
On Macros

By Paul Almond

3 April 2010

Website:

<http://www.paul-almond.com>

E-mail:

info@paul-almond.com

Software users often have requirements that are not met by the software. One solution to this is macros – external scripts that can interact with programs to provide extra functionality – but such a solution has shortcomings. In some environments, macros do not have logical access to the user interface of programs. Even when this is available, macros typically provide extra functionality that is not integrated with the user interfaces of programs. An approach is proposed which deals with these issues. Programs have two distinct user interfaces: a human user interface and a logical user interface. Macros can interact with the logical user interface of the program. Elements can be added to the human user interface of a program without any modification of its internal processing, the added elements interacting only with external macros, and this can make the functionality of a program appear to be extended. The extra functionality required in programs may be provided by adapting computer language software so that any programs produced with it automatically have the capability of working with macros in this way.

Table of Contents

1 Introduction	4
2 The Problems	5
2.1 Extending the Use of Programs	5
2.2 Macros: A Solution?	5
2.3 A Logically Accessible User Interface	6
2.4 What if there is already a logically accessible user interface?	7
3 Fixing the Problems.....	9
3.1 The Ideal Situation	9
3.1.1 User Experience	9
3.1.2 Programmer Experience	12
3.2 Clarification	13
3.3 Implementation	15
3.3.1 An Overview of How Implementation Could Be Done	15
3.3.2 Example 1: Personal Computers with “Conventional” Programs.....	16
3.3.3 Example 2: Web Based Systems Using Web Browsers as User Interfaces	17
3.4 Why does the logical user interface need to mirror the human user interface? ...	18
3.5 Examples	19
3.5.1 Example 1: Educational Software	19
3.5.2 Example 2: Extension of Customer Relationship Management Software	20
3.5.3 Example 3: Extension of Online E-Mail	21
3.5.4 Example 4: CRM Phone Pop-up	21
3.5.5 Example 5: Integration of Sales and Production	21
3.5.6 Example 6: Integration of Offline CRM and Online Appointments System	22
3.6 Languages, Interfaces and Standardization	22
3.7 Ownership Issues	24
3.8 Adding to the User Interface Standard	25
4 Conclusion.....	26

List of Abbreviations

CRM customer relationship management

GUI graphical user interface

HTML Hyper Text Markup Language

PC personal computer

XML Extensible Markup Language

1 Introduction

A computer program is commercially produced typically to satisfy many users. These users will tend to have varying needs, so many programs have many features, to try to satisfy all users. This may not be enough, and many programs can be customized significantly. This does not always make programs perfect for everyone. Individual users can have such unique requirements that just trying to design the software for them is impractical. An example of this is the requirement for software integration between two programs. Different users may want different combinations of programs to work together, with information being transferred between them, and there may be too many possible combinations for programmers to anticipate all of them.

One solution to all this is macros – scripts which can automate tasks involving programs – but these have limitations. In many environments, macros do not have the logical view of the program's user interface needed to work with it properly. If this is available (as it is in some environments) then there is the further problem that macros add functionality to a program, but are not properly integrated with its user interface.

In this article I will be showing how these problems can be resolved.

2 The Problems

2.1 Extending the Use of Programs

Suppose you have some repetitive and time consuming task to perform with a computer's software. It may be some task involving a single program, for which the designers could have provided functionality, but did not. It may be some task involving multiple programs; for example, transferring information from one program to another. You might be able to hire a programmer to write code to do this, but this is expensive, and may be impractical, and the alternative is a lot of tedious typing. When we obtain software, we are stuck with what the designers of the software provided.

2.2 Macros: A Solution?

A macro is a script, encoded in a special macro language, which automates what a user might do. It describes a set of operations to be performed using the user interfaces of one or more programs. Macro scripts might be written like computer programs, or they may be made automatically – by a user “showing” the computer the steps, which are then recorded and turned into a script. Macro systems take two main forms:

- One type of macro system relies on special software, running independently of the programs on which it is going to work, to provide macros. The program will typically allow the user to “record” sequences of mouse pointer movements and keystrokes, the macro software then generating a script to perform these operations. Scripts might be reasonably sophisticated. For example, a script may copy some text into the keyboard where it is available to the macro script as a variable and can be used in various operations or decisions.¹ An advantage of a system like this is that it can be made available for practically any program on a system, because practically any program can be accessed with things like mouse pointer movements and keystrokes. A disadvantage is something that may be less obvious. Such a program, running outside the application programs on which it operates, does not have “proper” access to the user interfaces on which it is operating. The user interfaces on which it works are not “logically accessible” to it, and it has to interact with the application programs in a rather crude way.²
- The other type of macro system is built into the application programs on which it is supposed to be used.³ The advantage of this is that, because the macro language is built into the program, the user interface of the program is logically

¹ An example of a program like this is *Macro Express*, a commercially available program produced by Insight Software Solutions Inc., <http://www.macros.com>.

² This is not the fault of the people who develop these programs. It is a limitation imposed on them by the user interface not being logically accessible.

³ An example of this is the extensive macro language provided in *Microsoft Office*. The macro language can be used to write scripts which automate various tasks that a user may want to do in Microsoft Office.

accessible to a macro. A macro can give instructions to the program, and make enquiries of it, in a format that it can understand. (This concept of a “logically accessible user interface” will be discussed a bit more shortly.) The disadvantage is that, unless it is built into the program, this kind of macro system is unlikely to be available in many environments.

Current technology often presents us, then, with a choice: We can have a logically accessible user interface or we can use a macro language with many different programs, but we often cannot have both. One requirement, therefore, is to have a logically accessible user interface.

2.3 A Logically Accessible User Interface

Above, I was just talking about the idea of a logically accessible user interface. This is important to this article, so I will discuss this a bit more to ensure that I have been clear. (If you think that what I have already said was clear enough, you may wish to skip this, and go to the next section, 2.4, *What if there is already a logically accessible user interface?*, on Page 7.)

Suppose you were going to tell a human how to do some task using one or more application programs. You would specify the task in terms of operations performed using the user interfaces, and observations made of things going on in them. You would assume that the person knows what is going on with the user interface. For example, we might specify operations as follows:

- “After doing that, click on the OK button.”
- “Go to the File menu. Pull the menu down. If the Save As option is available, select it, but if it is grayed-out...”
- “Right click on the icon for the image. Find out its width and height. Go to the website back end program. In the record for this product, enter the width and height in the relevant fields.”

In the first example, you are assuming (reasonably) that the person can see where the OK button is, and you are making similar kinds of assumptions with the second and third examples too. The instructions can reference concepts associated with the program, both in terms of doing things and observing things, because the human user is able to make sense of the user interface.

If the user interface of an application program is being used by another program, maybe using a macro script, and the user interface is not logically accessible, then the instructions cannot be like the ones described above. Instead they are likely to take a form such as:

- “Move the mouse pointer to (250,45). Press the left mouse button. Move the mouse pointer to (200,241). Click the left mouse button. Press the SHIFT key and hold it down. Press the END key. Release the SHIFT key. Press CTRL-C. If the text in the clipboard contains the sequence “052”, move the mouse pointer to (605,745) and press the left mouse button. Then wait 2 seconds and move the mouse pointer to position (10,10) and press...”

The macro does not know where things like buttons, menus or data entry fields are in the target program. Its ability to find out what the target program is doing is limited as well. A human user may easily be able to inspect a list of customer records that the target program is showing in a list box, but the macro may be unable to get at this data. It is likely to have only a rudimentary ability to extract information from the target program; for example, by selecting text and copying it into the clipboard, where it can access it logically.

Not having the user interface logically available to macros means that the task of constructing the macro will be harder. A macro might be made by “recording” a human user going through the steps, but trying to make a macro by coding it, or altering the existing code of a macro would be an exhausting experience. This will limit the sophistication of macros that can be made, as making macros by recording user actions limits them to a single, “straight through” course of action, rather than involving significant changes based on decisions. Macros will be comparatively simple. Some information, that would easily be available to a human user just by glancing at the display screen, will be inaccessible to a macro, because there is no mechanism for getting it.

Having a logically accessible user interface is important to the idea of getting macros to work properly.

2.4 What if there is already a logically accessible user interface?

Some readers may think of programs that are made available to users over the Internet on websites. The World Wide Web is an environment that happens to deal with the issues just discussed: It actually does provide a logically accessible user interface, because web pages are delivered to a computer as text in Hyper Text Markup Language (HTML). With this kind of environment it is possible to write programs that can read a website and submit data to it, to access a website automatically. In fact, there are

programs that take advantage of the logical accessibility of website applications to provide website automation. However, something is still missing.

When a program is developed, the user interface is typically optimized to give the best possible experience for the user. When a user finds one or more programs inadequate for his/her purposes, he/she may create a macro. With a website automation program, such a macro may run on one or more websites. The problem is that it is *added on*. The integration between a macro and the systems on which it operates is weak. The user interface of the system on which the macro is operating remains the same.

Web service standards do exist to allow systems to interact with each other, and standards such as Extensible Markup Language (XML) exist, but these do not in themselves provide all the functionality that will be proposed here. They may, however, play a part in some implementation of the proposal. An important aspect of what will be proposed here is the ease of automating tasks. Various ways of achieving the access to program user interface's that will be described here may be possible, but for it to be practical it must be reasonably easy to write code to do this.

3 Fixing the Problems

3.1 The Ideal Situation

I will describe the ideal situation in terms of *user experience* and *programmer experience*.

3.1.1 User Experience

- The user interfaces of all application programs are logically accessible to macros in a macro language, or languages. A macro script can be written in such a language which interacts with the user interface of any program, or more than one program, in a deep way. (The user interface elements with which a macro interacts can be considered to be part of its own user interface. This means that elements of the user interface of a macro might really be in the user interface of another program.)
- Any program has two user interfaces: a *human user interface*, which is experienced by human users, and a *logical user interface*, which is accessible to macros.⁴ By default they are the same, in the sense that there is a one-to-one mapping between elements in the human user interface and elements in the logical user interface. For example, a data entry field which is visible on-screen in the human user interface has a corresponding logically accessible version in the logical user interface. Events which affect one user interface affect the other. For example, if a macro populates a data entry field in the logical user interface of a program with data, this would be visible to a human user in the human user interface.
- The protocols for accessing a program's logical user interface are standardized, at least within the same environment, and are as similar as possible across environments.
- Macros can be made by recording user actions, but much more powerful macros can be made by coding them, or modifying the code for recorded macros. A macro language is Turing equivalent: Any conceivable algorithm can be expressed in it.
- The ways in which macros access the logical user interfaces of programs are standardized for all programs. Macros can manipulate the logical user interfaces of programs (for example, selecting items from menus or clicking on buttons)

⁴ I am assuming two user interfaces: one for us and one for computers. This might be considered a special case. We might regard the general case as being n user interfaces, each belonging to a class of user, a user being a human or a program; however, I will ignore this here, as it is not important to the main idea.

and extract information from them (for example, by reading the contents of data entry fields). For any sequence of operations by a human user with a human interface, there is an equivalent sequence of operations by a macro with the logical user interface, which has the same result.

- A macro can contain code that calls another macro as a subroutine. A macro can return results, and when it is called from another macro, these are available to the calling macro.
- A macro does not just have to run through some sequence of steps and then stop. It can remain running, continually interacting with the logical user interface of one or more programs and reacting to events that may happen. For example, a macro may monitor the logical user interface of a particular application program, and take some action if the user enters a date earlier than 1 January 2005 in a particular field.⁵ A macro can be associated with one or more programs, so that it automatically runs while any of these programs are running.
- Macros can be stored and activated by a user when desired; however, user interaction with macros is not restricted to this. A conventional user interface can be specified for a macro, so that the user can interact with it like any other application program. For example, the macro may have windows, menus, controls, data entry fields, etc. Such a user interface consists of a human user interface and a logical user interface – logically accessible to other macros, in the standard way – just like any application program. This means that one macro can interact with the user interface of another macro just as it would with any normal application program.⁶
- Elements of the user interface of a macro can be added to the human user interface of another application program and its logical user interface. For example, if a macro is going to be applied to a particular program, then buttons, list boxes, data entry fields, etc. can be added to particular windows in that program, or pull-down menus could be added to it. It is important that what this means is understood properly. It may seem that I am saying that if someone writes a program, then other people should have the ability to *alter* the code. This is not true. The extra components added to the user interface of a program only have the *appearance* of being associated with the program. In reality, they are associated with macros. For example, if an OK button is added to a window of a program, when a user clicks on that button the program itself is not informed of this. Instead, one or more macros would be informed: The event is being dealt with outside the program. From the user's point of view, however,

⁵ The user would enter it into the *human* user interface, of course.

⁶ This should make sense, because I have already said that a macro can have its user interface elements in different programs: All this really means is collecting them in one place.

he/she is clicking on a button in that program. If a data entry field is added to a program's window, then only macros external to the program can read the contents of that field or put data in it: The program itself has no interaction with it. We are just using macros to *build on top of* other programs. Elements of the human user interface and/or the logical user interface that are not part of the program itself can be removed: This is just a reversal of what has just been discussed.

- Elements of a program's human user interface that are built into the program can be removed, but this does not change the logical user interface. Anything removed in this way remains logically accessible to macros: It is just not evident to *human* users. For example, a program designed to quiz students in a school may ask a student to enter his/her name in a data entry field, this data then being used to address the student personally. It would be possible to remove this field from the *human* user interface of the program, but the field would still exist in the *logical* user interface, and when a student starts to use the program the field could be populated with data from another program. This does not just have to apply to data entry fields: It could apply to entire windows. A program may have a window that pops up and which is apparently "removed" from the program completely: In reality, the window still exists logically and macros can interact with it, possibly entering data into it that is of no interest to humans.
- The human user interface of a program can be adjusted to such a degree as is practical. I am merely stating the desirability of this: How far it goes depends on what, if anything, can reasonably be achieved in the environment in which programs run. For example, it may be possible to rearrange the positions of elements of the human user interface in a window, or to change the size of a window. This may be useful to make room for elements that may need adding to the human user interface.
- The human user interface of a program can be suppressed while running a particular macro. This is just a special case of the above in which we may want a particular macro to do something invisibly with a program, with no evidence that the program is running, or that some operation is occurring.
- A macro can itself make changes to the human user interface of a program, or suppress parts of it while the macro is running. For example, a macro may click on a button in the logical user interface which causes a window to open, but may suppress the appearance of that window in the human user interface, so that it can perform some task unobtrusively. As well as unobtrusiveness, one advantage of the human user interface not having to reflect all macro actions is speed: A macro may be designed to perform millions of operations on some program, which would take longer if all this had to be reflected in the human user interface.

- Where any of the above changes to the user interface of a program (e.g. adding or removing elements from the user interface, suppressing parts of it, etc.) are made by human users, the way in which this is done is standardized for all programs in the same environment.
- Where any of the above changes to the user interface of a program are made by a macro script (e.g. a macro suppressing the human user interface of a program from showing some operation), the protocol that the macro needs to follow is standardized for all programs in the same environment.
- A database system should be available to macros: Designers of macros should be able to define databases to be used with them. This gives macros the capability to store, retrieve and manage their own data, independently of the programs on top of which they are built. It should be noted that this would be possible anyway with any decent macro language: An external database could be installed on the system and commands used to access it as an external system. Failing that, a program used to access the database system could be acted on by macros just like any other program. However, there is a case for database provision to be closely integrated with the macro system, so that the designers of macros can set up databases to work with their macros quickly. The reason for this is that database functionality has the potential to make macros much more powerful, as examples will show later.⁷
- The above features should be easy to use, so that macros can be written without great difficulty.

3.1.2 Programmer Experience

- The programmer does not have to do any significant extra work to have the above implemented in his/her programs. It is an automatic feature of the systems used to create programs and/or the environments in which they run. For example, the programmer does not have to worry about how to make the logical user interface work, and how macros will run to work on it.
- By default, the human user interface and the logical user interface are the same. Normally, when the programmer specifies an element of the human user interface, it will be automatically added to the logical user interface. In general, any element in the human user interface should be in the logical interface as well, as this maximizes the scope for macros to interact with the program.
- A programmer may specify an element in the logical user interface, but choose not to have it in the human user interface. The programmer would do this if he/she wanted to add some functionality especially for macros to use, which

⁷ 3.5: Examples on Page 4.

humans will not want to use, and did not want to “clutter” the human user interface.

- The programmer may be allowed to choose to have elements in the human user interface, while omitting them from the logical user interface. An explicit decision by the programmer should be required to do this, because this will reduce the capabilities of macros interacting with the program while adding no functionality. A programmer may also be allowed to limit the capability of macros to act on his/her program in other ways. For example, he/she may choose not to allow user interface elements associated with macros to be added to the human user interface, or parts of the human user interface to be removed or altered in other ways.

3.2 Clarification

Clarification is needed on one of the requirements given above. I stated:

“Elements of the user interface of a macro can be added to the human user interface of another application program and its logical user interface.”

I should explain what this means. Actually, this is one way in which things could be done.

Suppose a macro needs to interact with a data entry field and an OK button that already exists in some program, which we call *Program A*. The data entry field and the OK button exist in both the human user interface and the logical user interface of Program A. The macro access the data entry field and button in the logical user interface, and by doing this it manipulates Program A, as if text had been entered and the button clicked on using the human user interface. Also, if a human enters text into the data entry field and then clicks on the OK button, the macro can obtain the text and detect the click on the button, again using the program’s logical user interface.

Now, suppose there is another program, Program B, in a particular window of which we want there to be the appearance of a data entry field and OK button, even though there is neither. We want a user to be able to enter text into a data entry field, and click on an OK button, in what appears to be Program B’s human user interface. The required user interface elements (the data entry field and the OK button) are added to the human user interface of Program B, in the relevant window. For now, we will also assume that the data entry field and the OK button are added to the logical user interface of Program B. These appear to be part of Program B, but are not: There is no connection between what happens with these user interface elements and any of the processing in Program B. Instead, when a user enters text into the data entry field and clicks on the OK button, it is a macro that accesses these elements in the logical user interface of Program B to read the text and detect the button click. Access to these elements is not restricted to one macro: Any number of macros could access them. From the point of view of macros, these are just like any other elements of Program B’s user interface: It just happens that they are not “connected” to anything inside Program B.

We might ask if the data entry field and text button actually need to be in the logical user interface of Program B at all. As they are not accessed by Program B's code, it would be possible for them to be in the human user interface of Program B, but not in its logical user interface. They might be logically associated with some macro, or they may exist as entities separate from any program or macro. The distinctions between these possibilities are really just conceptual. All it affects is the syntax of how user interface elements added to a program are referenced in a macro's code.

When we add a user interface element to the human user interface of Program B, we may also add it to the *logical* user interface of Program B, so that the code in a macro to copy text from the data entry field into a variable has to reference Program B explicitly and looks something like:

```
CustomerName = ProgramB.SomeWindow.DataEntry18
```

or we may logically associate the data entry field with the macro itself, so that the code may look something like:

```
CustomerName = ThisMacro.DataEntry1
```

and even though the macro's code is not referencing Program B, DataEntry1 is still actually appearing in Program B. We would still regard the human user interface version of the data entry field as being associated with Program B, though this is really just conceptual. (If we do this we may allow other macros to access it, even though it "belongs" to this macro.)

Alternatively, we may make the logical version of the data entry field independent of both Program B and any macros using it, so that it can be logically accessed without any reference to a particular program or a macro. We might associate it with some data structure, so that the code in a macro to access it looks something like:

```
CustomerName = GlobalUserInterfaceElements.DataEntry1
```

and the human user interface version of the data entry field would still be in Program B.

Not having to reference specific programs or macros in code may have the advantage that the user interface elements can be changed – they can be moved from one program to another, or from one position in a program's user interface to another position – without the macro's code having to be changed (though such a change might be done automatically when the items are moved). It may be best, however, to allow more than one way of referencing user interface elements.

The issue of macro code having to change if the user interface elements change should be considered more generally, not just for user interface elements added to a program.

⁸ No attempt is being made here to write very realistic code, in any language in particular. I am just trying to give an idea of how things could be referenced in different ways.

We may wish to write a macro that does not depend on a particular program, so that it can be used with different programs as required. For this reason, it may be useful to have the ability to reference any user interface element in a macro's code without explicitly stating the program, to which it belongs. Instead, some identifier would be used, and that identifier would be mapped onto an element of a program's logical user interface outside the macro's code, so that a macro does not need to be application specific.

3.3 Implementation

3.3.1 An Overview of How Implementation Could Be Done

We now need to look at how this would work. I am suggesting the concept here, rather than details, and exactly how it is implemented would depend on the environment; however, I will give an idea of how it may be implemented in two kinds of system: one in which the user interface is not already logically accessible and one in which it is.

Few programmers will want to design all the functionality described above into a program: Programmers' desires aside, it would be economically unviable to build a system like this every time a program was created. If the functionality is made available by things outside the program – by the environment in which the program runs, and possibly the tools used to create the program - everything becomes much more feasible. An analogy would be graphical user interfaces (GUIs). It would be impractical to provide programs with the GUIs that they have if a lot of this functionality were not provided by the environment in which programs are created and run. This is why I said previously that the ideal is for this kind of functionality to be in programs automatically, without programmers having to build this kind of functionality themselves. Any other solution is impractical.

What is needed to provide this functionality depends on the environment in which the programs are running. With some environments, the tools used to create or run the programs need to be adapted to provide at least some of the functionality. In other environments, all of the functionality can be provided, if desired, by adding systems outside the programs on which macros are to act – leaving the programs' internal code untouched.

The most basic requirement is for each program to have a logically accessible user interface – what I have described as the logical user interface. If this is available, a system outside the program can, in principle, interact with it and this can be used to provide the rest of the system. If programs do not have logically accessible user interfaces then this needs to be fixed before anything else can be done and this means adapting the tools used to create or run the programs. With compiled programs it will tend to mean adapting language compilers so that a logical user interface is automatically provided with any program produced with them. With interpreted

programs this will tend to mean adapting language interpreters to provide the required functionality automatically when running programs.

The next requirement is for a human user interface which is distinct from the logical user interface. With some environments this can be provided by systems in the environment, but in other environments it will be more practical to build it into the language tools, as with the logical user interface functionality. In general, if the language software has to be altered to provide a logical user interface it will need altering to provide a distinct human user interface.

Once a distinct logical user interface and human user interface are available, we need to get macros executed.

We could provide the macro functionality by adding extra, special software outside the programs to be acted on. This would mean that the act of creating and running a macro would involve some other system which acts as an intermediary: The programs on which the macros are acting do not need to know how to deal with macros, but instead this external system which is dealing with the macros would be sending requests for action and information to the logical user interface. This would allow different languages to be used as the macro language to access a program's interface: The program on which the macros are operating would not be specific to any macro language, making this approach desirable. Existing languages could then be adapted for use as macro languages – possibly with add-on modules, or possibly just with calls to external services.

We may choose to adapt computer language software to provide all the required functionality, so that any computer program produced with (or interpreted by) some programming language package automatically has the capability to provide execution of macro scripts that access its user interfaces, and possibly those of other programs. This would mean that creating a macro, and running it on one or more application programs, would involve interaction with these programs themselves only: Nothing else would have to act as an intermediary, as the programs themselves would “understand” how macros work. If this is done, however, there should still be the facility to run macros externally to the program, as described above, with the program just responding to instructions and requests for information sent to its interfaces, to avoid forcing a particular macro language onto users.

I will now look at two examples:

3.3.2 Example 1: Personal Computers with “Conventional” Programs

This situation is the one typical on personal computers (PCs) at the time of writing. Programs on computers are typically written in high level languages using language software (such as *Microsoft Visual Basic*) and either object code produced by a compiler

is installed on computers or code is interpreted.⁹ Some programs are designed to provide macro functionality, but most are not.

In this kind of system, there tends not to be an existing logical user interface. This means that just placing software on the system to provide the required functionality from outside programs is not practical in most situations. The language software used to produce object code or interpret programs while they are running needs to be altered to provide a logical user interface. The functionality of the human user interface falls short of what is required. Some of the required functionality may be achieved by systems working from outside programs, but it would be limited, and it would be complex and inefficient to do this. (As an example, we might imagine a program overlaying an extra display on top of a program's windows.) As we have to alter the language software anyway, it makes sense to alter it to provide the required functionality for both the human and logical user interfaces.

With this functionality achieved, there is the issue of getting macros executed. This could be done by specific software added to computers for this purpose, which acts as an intermediary between a macro script and the user interfaces of the programs on which it is operating: Such a program could be added to the system like any other program, or built into the operating system. Alternatively, the alterations to the language software mentioned above could also include making computer language software provide the full macro functionality with any programs produced or executed using them, or for the language software to enable programs to do *some* of the work involved in executing macros that operate on them, the rest being done by software outside these programs. However, if a programming language automatically makes provision for macros to be used with programs produced with it (or interpreted by it), it should still be possible to use other programming languages to access them: The user interfaces of the program should always be accessible to whatever language the user wants to use, regardless of what the program itself might be made to recognize.

3.3.3 Example 2: Web Based Systems Using Web Browsers as User Interfaces

Many people now access applications over the Internet, using a web browser as a user interface. As described previously, this already provides a logical user interface; however functionality is required for the rest of the proposal being made here. The following are some obvious ways of doing this:

One way involves the software which provides the required functionality being added to the user's computer. This software would need to be able to read the contents of pages in a web browser, as well as accessing input fields on forms, but all this can already be

⁹ An approach intermediate between these two approaches is sometimes used, involving partial compilation followed by interpretation.

done. The proposal calls for a program's human user interface to be altered without interfering with the program itself. In the context of web based software, this means altering what is provided by the web browser without altering the software on the server, something that is clearly possible. The software which provides this functionality could do this by locally altering the HTML code stored in the browser, to present a different human user interface – one with the required changes. The proposal calls for a database being made available to macros, and this could be on the user's computer or on the Internet.

Another way involves using a web browser with the required functionality built into it. It would interact with application programs – websites – over the Internet, adding what is required to be built over the top of them by macros before presenting them to the user.

Another way involves putting some of the system that provides the macro functionality onto the Internet. The user may access a website which runs various macros that access other websites and presents the human user interface required by its macros to the user. A system like this could allow a macro to be made available for many users. It should be noted that the human user interface presented to the user in this way does not have to be “built from scratch” by the macro: It may just be an existing web page with a button or input field added to it, or it may be an entire user interface specific to the macro.

Other solutions may involve websites providing a completely separate logical user interface optimized for access by programs, possibly using an existing web service standard, but if this is done the programmer should not have to create this interface: The human user interface should be automatically mapped onto it. As with the previous example, the tools used to create applications programs (in this case the programs on websites), or run them, could be modified so that they automatically provide this capability. If such an approach is used, however, a special logical user interface should not be mandatory: It should be assumed that most websites will just have a standard HTML user interface. The system running the macros might be able to detect the existence of an optimized logical user interface automatically, and use it if it is there.

3.4 Why does the logical user interface need to mirror the human user interface?

Some readers may wonder why I am so concerned with having a logical user interface that mirrors the human interface. Why would I want macros to access programs in the same way as humans? There are two reasons for this:

- It is the only practical way of giving macros the capability of doing the same things as humans. Most programs have a human user interface, so the functionality to do this needs to be developed anyway. Having the functionality automatically mirrored in the logical user interface allows all the work of

developing a human user interface to be used in the logical user interface as well. The alternative would be for programmers to have to develop both systems separately. This would be uneconomical and corners would be cut. Users writing macros would constantly be finding that some task they wanted to do, and which was possible in the human user interface, was impossible in the logical user interface, and so could not be automated.

- Making the logical user interface like the human user interface makes it close to the experience of human users. Users will tend to be familiar with the human user interface, and will find it easy to work out how to express a task as a sequence of operations in the human user interface, and then map it onto the logical user interface. A user can examine the human user interface at any time to determine what operations are needed to accomplish a goal – and the documentation for the human user interface also serves as logical user interface documentation. A macro can be played back and watched by a human user as it performs its task. This is preferable to a user having to first learn the human user interface for the software and then study a macro writing guide describing all the complicated methods of getting macros to interact with the software.

It is worth pointing out that it is not compulsory for the logical user interface to be the same as the human user interface. This is simply the minimum functionality that the logical user interface provides, and it is provided by default. A programmer can add extra items to the logical user interface if he/she wishes.

3.5 Examples

Some examples now follow of the kinds of things that could be done with a system like the one proposed.

3.5.1 Example 1: Educational Software

A teacher is using a teaching program installed on a PC with a class of students. The program teaches students and gives them quizzes. The teaching program has not been designed to store the scores of different students: In fact, there is no way of telling it which student is using it. The teacher wants this functionality. The teacher writes a macro and adds features to the human user interface of the teaching program to work with this macro. Now, when a student starts to use the program, it asks him/her to enter login details into data entry fields that appear to be in the teaching program. The program then addresses him/her by name and the student's results for the quizzes are stored in a record. The teacher can log into the system to see the results for different students.

In reality, what may seem to be a modification of the teaching program is merely extra functioning built on top of it. The data entry fields to log in to the teaching program are merely added to its human user interface for use by a macro, and the program itself cannot even "see" them. When a student logs in to the teaching program, it is a macro

that picks up the entered data. When the student completes a quiz, the macro obtains the result from the logical user interface of teaching program. The macro then accesses a database which is managed by the macro, and not by the teaching program at all, and stores the quiz result in the student's record.

This example should have shown why I regarded it as important for macros to have access to a database. A macro does not have access to the program's own data, but by accessing a database outside this program, it can be made to appear as though it is working on data stored within the program.

3.5.2 Example 2: Extension of Customer Relationship Management Software

A program used to access a customer relationship management (CRM) database deals with customer records. Each record stores details for a customer, such as name, telephone number, address, etc. There is no facility to store comments as part of the customer record, and this is wanted.¹⁰

An element to display the comments is added to the CRM program's human user interface, in the window used to display a customer record. A macro has access to this. When a customer's record is displayed, a user can enter a comment to be added to the customer record. The program then appears to store it as part of the customer record, so that the next time that customer record is displayed, the comment is displayed with any previously entered comments for that customer: It appears to be in the same database as the rest of the CRM data. What is really happening is that the macro is managing a separate database. When a user adds a comment to a customer record, the macro observes this. It accesses one or more data entry fields in the CRM program's logical user interface to obtain enough information to uniquely identify the customer. It uses this information to locate a corresponding customer record in its own, separate database – creating the record if it does not exist. Later, when the customer record is displayed the macro obtains the information identifying the customer from the CRM program's logical user interface. It then uses this to look up the relevant record in its own database again, obtains the comments from this database, and adds these to the element used to display the comments in the human user interface of the CRM program.

This is another example showing the utility of a database capability with macros. In this case, a second database, managed by a macro outside a program, is made to appear to the user to be part of the database managed by the program: It allows us to do things that could otherwise be done only by modifying the program's associated database.

¹⁰ I know that this functionality would normally be expected in a CRM system. This is just an example of what could be done, though.

3.5.3 Example 3: Extension of Online E-Mail

A website provides web mail facilities and is used by a company's sales representatives. When an e-mail from a user is received, a photograph of the user, a map of his/her location, the local time where the user is and notes made by various people in the company is displayed. This appears to be part of the website: The HTML to display these appears to be on the website in the appropriate places. In reality, a macro is dealing with this, with the human user interface presented to the sales representatives being altered accordingly. The macro is using a separate database to store this information, and it is extracting information identifying people from web pages to determine which records to look up in its own database.

3.5.4 Example 4: CRM Phone Pop-up

A small business has its phone system linked to its computer network. When someone telephones the company, call-logging software on its computer network logs the number. The company also has a CRM system. They want to arrange it so that when an incoming call is received, if the number belongs to anyone stored in the CRM database, the relevant customer record automatically opens in a pop-up window. The problem is that the two systems are not designed to work together.

This problem is solved using a macro. When the phone rings, a macro obtains the phone number from the logical user interface of the call-logging software. It then access the logical user interface of the CRM software, using the elements in it that would be used in the human user interface to search for a customer with the known phone number and display the record, if found.

3.5.5 Example 5: Integration of Sales and Production

A company is using web-based systems to manage information. Two systems are used, each having a different website. One system deals with sales orders. Another deals with arranging manufacture of custom products. They want to link both systems together, so that sales orders are used to instruct manufacture of custom products.

A macro obtains sales order information from the logical user interface of the sales order system and inputs it into the manufacturing system, using the logical user interface. Further information may be transferred between the two systems during the progression of an order and the associated manufacturing.

Elements are added to the user interface of the sales order system, on the web page used to display a sales order, to display a summary of manufacturing status, and allow a user to authorize manufacture for an order. There is also an option to go to the manufacturing system itself to view the manufacturing details for any given order. Elements are added to the user interface of the manufacturing system, on the web page used to display manufacturing details for any given sales order, to display a summary of the sales order information. There is also an option to go to the sales order system to

view the relevant sales order. All of this is done by building on top of each system, and these added user interface features are interacting with a macro, rather than with the sales order or manufacturing themselves, but to a user it appears that both systems have been extensively integrated – that someone has worked on the code.

3.5.6 Example 6: Integration of Offline CRM and Online Appointments System

A small business uses an offline CRM system, installed on its own computers, for managing contacts. They also use an appointments system on a website. Their customers can use this website to book appointments, and their own staff can then access this using an administrator login on the website. They want the two systems to be integrated. They cannot modify either system. The CRM software is a commercially bought package, and the appointments booking website is owned by another company: They pay a monthly fee to use it.

A macro accesses the logical user interface of the website and obtains customers' appointment information. It stores this in a database, separate to the data stored by either the CRM system or the appointments system. Elements are added to the user interface of the CRM system, in the window used to display a customer record, to show appointments information, and when a customer record is displayed the macro causes this information to be displayed with the customer record in the CRM system. A link to the appointments system is also added to the CRM customer details: Clicking on this link for a customer in the CRM goes to the web page displaying that customer's appointment details. All of this is managed externally to either system, by the macro.

This last example involved a macro system that could access systems in two different environments: the environment of programs installed on a computer locally and the World Wide Web. Such a macro system would be the obvious progression of the ideas proposed here, for maximum functionality. Initially, different systems could be developed to provide the proposed functionality in different environments. For example, a system to work with programs installed on computers might be developed by modifying programming languages and another system might be developed which extends the functionality of existing web automation systems to provide it on the World Wide Web. The system would become really powerful, however, when these different systems became combined, so that practically anything could be integrated with anything else, with user interfaces being built on top of the combination as required.

3.6 Languages, Interfaces and Standardization

It is not necessary for a single macro language to be used. Different languages could be used, provided that the logical user interfaces provided by all programs are standardized, at least within a given environment, so that the user interface of any program could be accessed using any macro language provided that programs created in that language met the required standard. This could mean that the syntax in the

macro language to access the logical user interface of a macro could vary from language to language, although the underlying format of the request made by the code would always be the same in a given environment. Although there may be differences from language to language, the reliance of it all, at a low level, on the same underlying protocol for communication with program user interfaces would tend to mean that the syntax for accessing program user interfaces would at least be *similar* in different languages.

A macro language used in an approach like this would need to be a general purpose language, and it could be used to produce programs with human interfaces. There is no reason why any language used should not be useful as a programming language for writing software in general, so that a “macro” would just be a special case program which happens to access other programs. This kind of general use should be considered when choosing a language to be used, and if the language is not suitable for this kind of general use, its usefulness for writing macros should be questioned. It may be desirable to adapt existing, general purpose programming languages so that they can be used to write macros, of course ensuring, if the environment requires it, that any programs produced using them comply with the logical user interface standard. This would be desirable, because it would mean that people could use their existing programming skills to write macros. In fact, the distinction between conventional programs and macros may start to become vague. Many existing languages could be adapted for use in writing macros.

A language may be adapted for use in writing macros by an add-on module. Alternatively, the actual access to the program being acted on by the macro might not be made by the macro directly, but by an intermediate program which is independent of the language being used. This would be similar to the way database drivers are used to access databases. We might call it a *macro driver* or *user interface driver* and it would accept instructions and requests to give to the program’s user interface in some syntax that is independent of the language in which the macro is written, returning the results to the macro. The programmer may need to use some language specific features, such as arrays, to process the returned data, but the standardization achieved could be significantly improved this way and it should allow the use of almost any language that already allows programs to use external systems such as database drivers.

To encourage widespread adoption of the proposed system, at least one language available for use in writing macros should be easy to learn and allow simple problems to be solved with minimal work: It should be a language in which a simple “Hello World” program really is simple. In general, use of the facilities described here in a programming language should be as easy as possible. If the system is provided by adapting programming languages so that programs written in those languages provide the logical user interface, as well as having the other functionality that is required, then a programming language may automatically allow the use of itself as a macro language to access any program written in that language. This would be most practical for

interpreted languages. If this is done, however, the program's interfaces should be available for use with other languages.

Although it is desirable for the logical user interface to be standardized, this does not need to happen immediately. Different standards for the logical user interface could be offered at the same time, and intermediate software could be used to allow communication between different systems. In the end, a single system, or a small number of systems, would become widely adopted.

3.7 Ownership Issues

An issue that needs consideration is that of ownership of the intellectual property in the software that is being built on like this. Software companies normally view modification of their programs negatively. Software licenses often specifically prohibit it. The software producer often keeps the source code private, while the user is only provided with the object code, which makes such modification difficult. Does this kind of issue arise with what I have been discussing here?

The two situations are not really the same though. In the proposal here, no changes are being made to the code of any of the programs. Instead, other systems are interacting with them, automating what a user might do. In principle, software companies could object, and could write licensing agreements appropriately – but *in principle* they could object to anything. We should consider whether or not software companies have any reason to object to what is, just a different way of using their products.

In the case of a commercially bought program, installed on a computer, there should be little reason for software companies to object. They make their money by selling utility to users, and programs that can be adapted to users' needs and can be made to work together have greater utility than programs that do not. We should expect them to *want* their programs to work with macros as described here, and to use programming language systems which provide the relevant functionality. A software company could choose not to allow this – and in the case of software installed on a PC this could easily be enforced just by making sure that the required functionality is not built into the software – but this would just mean that a competitor's program that could be made to work with macros could be adapted for uses for which the first program could not. It would not usually be in the interests of a software company to act like this.

On the other hand, while any general objection seems unlikely, there could be situations in which specific software companies object to specific things that people want to do with macros. A software company producing bespoke software may not want the task of integration to be so easy. A software company might not be happy if someone builds an entirely new application on top of theirs, so that their own application is reduced to the status of an invisible "cog". This objection is questionable: After all, anyone wanting to use the macro would have to buy the program anyway, but the software company

may feel that its products should always be visible. In the case of a freeware program, the programmer may regard the recognition he/she gets for writing it as the only payment he/she is receiving, and may expect the program to remain visible. Issues like this might be resolved by allowing software producers to specify limitations on macros that operate on them. Other issues could be thought of for software installed on people's computers, and I have pointed these out just to recognize that the issues exist. The point is that these are specific objections, against a general idea that could make many of these programs much more useful to the people buying them.

The issue of web applications may cause a bit more difficulty. Software on the web is often not paid for directly by users, but is supported by advertising. A web business supported by advertising would be likely to object if someone used their system as a building block for another system, without them gaining anything from it. As was suggested for software installed on computers, this could be dealt with by allowing software producers (in this case, web companies) to specify how much macros can do with their systems. For example, they may specify that a certain number of elements can be added to the human user interface, but that a completely new user interface cannot be created. They may allow a completely new user interface to be created, provided that it involves display of certain elements that they specify. They may allow their systems to be used if they are appropriately rewarded. Any compliance with such demands would need to be built into the macro system: The alternative would be for the web companies to start blocking access. It should be noted that nothing discussed here creates a completely new problem: Websites are already accessed by automated systems. What is different here is the scale on which this may happen.

3.8 Adding to the User Interface Standard

I have described a lot of this in terms of things like buttons and data entry, but it could be taken further. Ideally, provision would be made to add elements to the logical user interface protocol. This would allow more complicated user interface elements to be added over time, and new user interface elements to be added as they are developed.

An example would be interaction with graphics software. Protocols for how a program that allows the user to manipulate images can communicate with other programs would allow the parts of the user interfaces of those programs that deal directly with image manipulation to be built on. Ultimately, even a single pixel in an on-screen image might be a logically accessible user interface element.

4 Conclusion

Commercially produced software is generally designed for many users. Users often have specific requirements that are not met by the software. Macros provide one solution, by automating tasks that the user would otherwise have to perform him/herself, but they have limitations. In some environments, macros do not have logical access to the user interface of programs. In other environments, such as on the World Wide Web, with programs that have a web interface, such logical access to the user interface can be available, but there is a further limitation: Macros typically provide extra functionality that is not integrated with the user interfaces of programs. A way of resolving these problems has been proposed. This would involve developing a system that provides certain features in the user experience and the programmer experience.

The following would be ideal features of the user experience.

- The user interfaces of all application programs are logically accessible to macros in a macro language or languages.
- There are two distinct user interfaces – a logical user interface and a human user interface, though by default these will be the same.
- The protocols for accessing a program's logical user interface are standardized, at least within the same environment, and are as similar as possible across environments.
- Macro languages are Turing equivalent.
- The ways in which macros access the logical user interfaces of programs are standardized for all programs. For any sequence of operations by a human user with a human interface, there is an equivalent sequence of operations by a macro with the logical user interface, which has the same result.
- Macros can call macros, and a macro can return results when called.
- A macro can remain running continually, waiting for various events to occur. A macro can be associated with one or more programs, so that it automatically runs while any of these programs are running.
- Macros can have user interfaces – and these are treated as any other program's user interface, with a distinction between the logical user interface and the human user interface.
- Elements of the user interface of a macro can be added to the human user interface of another application program and/or its logical user interface. However, when this is done they appear, to a human user, to be part of the

- program they have been added to. In reality, no modification of the program's code has occurred, and these elements are just interacting with the macro.
- Elements of a program's human user interface that are built into the program can be removed, but this does not change the logical user interface. Anything removed in this way remains logically accessible to macros: It is just not evident to *human* users.
 - The human user interface of a program can be adjusted to such a degree as is practical. This may be useful to make room for elements that may need adding to the human user interface.
 - The human user interface of a program can be suppressed while running a particular macro
 - A macro can itself make changes to the human user interface of a program, or suppress parts of it while the macro is running.
 - Where any of the above changes to the user interface of a program (e.g. adding or removing elements from the user interface, suppressing parts of it, etc.) are made by human users, the way in which this is done is standardized for all programs in the same environment.
 - Where any of the above changes to the user interface of a program are made by a macro script (e.g. a macro suppressing the human user interface of a program from showing some operation), the protocol that the macro needs to follow is standardized for all programs in the same environment.
 - A database system should be easily available to macros.
 - The above features should be easy to use, so that macros can be written without great difficulty.

The following would be ideal features of the programmer experience.

- The above functionality is an automatic feature of the systems used to create programs and/or the environments in which they run, so that the programmer does not have to provide it explicitly.
- By default, the human user interface and the logical user interface are the same.
- A programmer may specify an element in the logical user interface, but choose not to have it in the human user interface.
- The programmer may be allowed to choose to have elements in the logical user interface, but not in the human user interface.

It is important that programmers are not required to do lots of work to provide this functionality in every program written. For software installed on PCs this probably means modifying computer language software so that the programs produced or interpreted by it automatically have the required functionality. This could provide a significant economy of scale: Adapting a single programming language package would automatically make everything produced with it compliant with the proposal described here, allowing any program produced with it work with macros, and with other programs produced with it. For software accessed through websites this functionality should already be available, to some degree, because of the way HTML pages are encoded and delivered to browsers.

Different languages could be used as macro languages, provided that the protocol for accessing the logical user interface is standardized. In general, using the macro facilities within a language to make a macro should be as simple as possible. Existing computer languages might be adapted for use as macro languages. If the system is provided by adapting existing programming language packages, a programming language package may automatically allow the use of itself as a macro language for accessing any programs made using it, although it would still be desirable for users to be able to use other languages. Access to the user interfaces of programs might be provided by an intermediate system, a macro driver or user interface driver, which a program can use as a service in much the same way that programs use database drivers.

Ideally, it would be possible to add to the user interface standard over time.

If the proposal made here became reality, software could be much more easily adapted to users. Software companies should like this because it means that their products will better match user's needs. In fact, it may even be a way for software companies to find new features to put in their programs: A user may create a macro that impresses a software company enough that they build the functionality into the program, where it can be provided more efficiently.

Regardless of whether this kind of proposal is considered worthwhile, one thing that we should stop doing, right now, is any more damage. Programs installed on modern PCs tend to have user interfaces that are obscured from other programs, making automation impractical. (Part of the proposal made here has been for fixing this.) This was not always the case. Before GUIs, when computers had to communicate with humans using command line interfaces, the user interface of a program was, in principle, more logically accessible, because it involved text that could be captured, processed and produced by programs. Historical systems, ironically, had greater scope for automation than the more sophisticated systems that now exist. The World Wide Web is an improvement in this respect, because by providing a logically accessible user interface, it repairs some of this damage. In any future developments that change user interface standards, the decision should never be taken to make a user interface less logically accessible, as has been done, with great cost, in the past.