<*name*>
<*College*>
<*CRSID*>

Diploma in Computer Science Project Proposal

A Ladder Logic Compiler and Interpreter

<*date*>

Project Originator: This is a Model Project Proposal

Resources Required: See attached Project Resource Form

Project Supervisor: <*name*>

Signature:

Director of Studies: <*name*>

Signature:

Overseers: <*name*> and <*name*>

Signatures: <*no need to obtain Overseers' signatures yourself*>

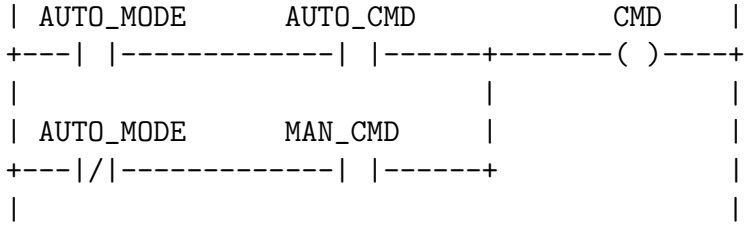
<This Model Proposal describes a project which involves the design and implementation of a language based on one used in programmable controllers. This document gives a summary of a possible design and suggests how it can be implemented. It is perfectly possible to use a different implementation language and to use different data structures. Likewise the way the simulator is implemented is simply given as an example. At various points in this document alternatives *are suggested or implied, and the submitted proposal is to be selected from the options given.>*

Introduction and Description of the Work

Ladder Logic is a simple notation used by engineers for controlling industrial processes. It can be used describe simple programmable controllers such as those used in traffic lights, lifts or fire safety systems. Full details of ladder logic can be found in Part 3 of standard IEC 1131 (available in the library as V111-26). This project is however concerned with a cut down variant of the language and a detailed understanding of the standard will not be necessary.

The concept of Ladder Logic is drawn from conventional relay-based logic, where relay & switch contacts are connected in series/parallel combinations to control the energisation of further relay coils, indicator lamps, solenoids etc. The relays may have built-in functions such as time delay, threshold detection etc., so the resulting circuit can achieve a reasonable degree of sophistication.

Ladder Logic is so called because the two *uprights* of the ladder are effectively the power supply and ground bus-bars, and each *rung* is (in the simplest form) a sequence of contacts in series/parallel which will energise a given coil. An example of a Ladder Diagram:



Where `--| |--` is a normally-open contact, `--|/|--` is a normally closed contact, and `--( )--` is a relay coil.

The above diagram translated into (un-optimised) C language would be:

```
CMD = (AUTO_MODE && AUTO_CMD) || (!AUTO_MODE && MAN_CMD);
```

or into pseudo assembly language it might be:

```
LOAD    AUTO_MODE
LOAD    AUTO_CMD
SYS     $AND
LOAD    AUTO_MODE
SYS     $NOT
LOAD    MAN_CMD
SYS     $AND
SYS     $OR
STORE   CMD
```

In the full Ladder Logic language, elements are not constrained to simple boolean operations, but can contain sophisticated arithmetic functions (e.g. auto-tuning PID control loops). The size of the project can be varied as the project progresses by adding more elements to the language but this should be done only if time permits.

It is tempting for computer professionals to dismiss Ladder Logic as a crude language, but it is a salutory thought that that almost every automated industrial process has one or more programmed logic controller running Ladder Logic, so we are unknowingly relying on Ladder Logic to provide many of the mass-produced items we use in our daily lives!

In conventional programming languages, the separation of the source text into source lines is simply an aid to human-readability. In a ladder diagram the vertical and horizontal orientation of the text elements is crucial, and directly effects the meaning of those elements. Hence the source code is effectively two-dimensional, as opposed to the one-dimensional (linear) source code of conventional languages. One key part of the project is to create a compiler that can handle such two-dimensional source code in a clean, well-structured fashion, which allows extra graphical elements to be added with minimal effort.

Resources Required

<The most suitable language may turn out to be non-standard and its availability needs to be established. The project can also be undertaken (with permission) on a privately owned PC.>

Starting Point

<This is the place to declare any prior knowledge relevant to the project. For example any relevant courses taken prior to the start of the Diploma year.>

Substance and Structure of the Project

The objective of the project is to design a ladder-logic style language and implement its compiler and interpreter/simulator so that the behaviour of a controller specified in the language can be simulated. The input of the compiler should be ASCII text to represent the two-dimensional logic diagram.

Notations for wire joining and crossing should be chosen, and also a method of entering relays with delayed action should be designed. The exact rules governing how names can be associated with relays and coils must be specified. The chosen ladder logic notation should then be tried out by specifying some simple control circuits. Some of these will be used later when the system is being tested.

Once the language details have been clarified, work can start on the compiler. This could consist of a syntax analyser that constructs an internal representation of logic diagram, followed by a translator that converts this into a form suitable for interpretation.

One possible representation of the circuit might be a list of relay nodes, a list of coil nodes and a list of wire nodes with suitable inter-node references to represent the circuit. For instance, a relay node could contain pointers to the activating coil and the two wires attached to its terminals. The syntax analyser will need some form of lexical analyser to handle low level input and deal with symbolic identifiers.

The compiler will contain a translation phase that converts this internal representation of the circuit into a form that is suitable for efficient interpretation. This might consist of a combination of interpretive code and pre-computed tables.

The interpreter must be capable of simulating the execution of the specified controller. It is likely that it should be written in the style of a discrete event simulator with a time queue of events to be processed in time order. A typically event might be the opening or closing of a relay at some specified time in the future. If this is the next event to happen, time will advance to that moment and the action will be performed. This may affect the signals on some wires which in turn may activate or deactivate some relay coils. This may result in a cascade of new events being sent to the time queue. During simulation user readable output should be generated so that the behaviour of the controller can be clearly understood.

Some ingenuity will be required to find a good way of determining which coils activations change when a given relay changes state. If there are fewer than about 16 relays this can be precomputed in the form of a table, but for larger circuits other strategies should be considered. A possible approach might be based on dynamic programming.

Variants

The entire project can be implemented in Java, C or C++ but other implementation languages could be considered.

Rather than using an ASCII representation of the logic, it would be possible to use a graphical representation constructed by a specially written graphical editor. This may, however, make the project too large for most candidates.

It may turn out to be a better idea for the simulator to run to completion generating an internal representation of the behaviour that can be inspected later by an interactive viewer. The viewer program should allow the user to decide what parts of the circuit and/or time period to view. If this scheme were adopted it would be necessary to design a good internal representation of the behaviour. Ease of generation, ease of use and compactness would be the main conflicting issues affecting the design. The viewer could generate its output in simple ASCII text, or it could generate simple pictures in ASCII using the "curses" library package under Unix, or it could generate more attractive graphical output using, for example, the graphical facilities of Java.

Success Criterion

For the project to be deemed a success the following items must be successfully completed.

1. A notation in the style of ladder logic needs to be designed and specified in detail.
2. An internal representation of the ladder logic program must be designed.
3. Program/programs must be designed and implemented to convert this notation into a form suitable for interpretation/simulation.
4. The data structures and algorithms used by the discrete event simulator must be designed and implemented.
5. Details of what output the system should generate and how it should be controlled (by the user) must be specified.
6. Demonstration test programs should be written and run to demonstrate the the project works.
7. Some indication of the efficiency of the implementation in terms of space and time should be given.
8. The dissertation must be planned and written.

<The sketches above are to be taken as starting points for investigation of the literature and discussion with your Supervisor. A look at the 1996 Diploma Dissertation, A Ladder Logic Compiler and Interpreter*, by Johnson Adesanya, is likely to prove useful since it provides some pointers into the literature, and shows in detail one way of solving the problem.>*

<It would also be possible to adjust the style and emphasis of the project either towards the best possible computational capability for your code, or in the direction of better human/machine interaction, particularly for the visualisation of the behaviour of given circuit designs.>

Timetable and Milestones

<In the following scheme, weeks are numbered so that the week starting on the day on which Project Proposals are handed in is Week 1. The year's timetable means that the deadline for submitting dissertations is in Week 34.>

<In the Project Proposal that you hand in, actual dates *should be used instead of week numbers and you should show how these dates relate to the periods in which lectures take place. Week 1 starts immediately after submission of the Project Proposal.>*

<The timetable and milestones given below refer to just one particular interpretation of this document. Even if you select exactly this interpretation you will need to review the suggested timetable and adjust the dates to allow as precisely as you can for the amount of programming and other related experience that you have at the start of the year. Take account of the dates you and your Supervisor will be working in Cambridge outside Lecture Term. Note that some candidates write the Introduction and Preparation chapters of their dissertations quite early in the year, while others will do all their writing in one burst near the end.>

Before Proposal submission

<This section will not appear in your Project Proposal.>

Submission of Phase 1 Report Form. Discussion with Overseers and Director of Studies. Allocation of and discussion with Project Supervisor, preliminary reading, choice of the variant on the project and language *<Java in this example>*, writing Project Proposal. Discussion with Supervisor to arrange a schedule of regular meetings for obtaining support during the course of the year.

Milestones: Phase 1 Report Form (on the Monday immediately following the main Briefing Lecture), then a Project Proposal complete with as realistic a timetable as possible, approval from Overseers and confirmed availability of any special resources needed. Signatures from Supervisor and Director of Studies.

Weeks 1 to 5

<Real work on the project starts here (as distinct from just work on the proposal). A significant problem for Diploma candidates is that this critical period largely coincides with the Christmas vacation. There is no guarantee that supervisors will be available outside Lecture Term, but Diploma students take much less of a Christmas break than undergraduates do, and so have some opportunity for uninterrupted reading and initial practical work at this stage. It is important to have completed some serious work on the project before the pressures of the Lent Term become all too apparent.>

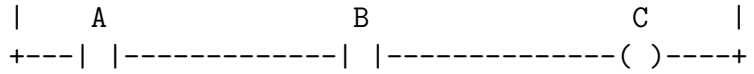
Study C and the particular implementation of it to be used. Practise writing various small programs, including key fragments of the compiler and interpreter.

Milestones: Some working example C programs including code to deal with symbol tables in the lexical analyser, and part implementation of the time queue to be used in the interpreter/simulator.

Weeks 6 and 7

Further literature study and discussion with Supervisor to ensure that the chosen data structures are satisfactory. Implementation of the syntax analyser and debugging code to help test it. This is likely to be code that can be used to display the data structures in human-readable form so that it is possible to check that they are as expected.

Milestones: Ability to construct and display data structures that represent simple ladder logic programs such as:



Weeks 8 to 10

Implementation of the translation phase of the compiler. This will of a decision to be made on what target code to generate. The obvious contnders are: an interpretive code, Java or C.

Start to plan the Dissertation, thinking ahead especially to the collection of examples and tests that will be used to demonstrate that the project has been a success.

Milestones: Ability to compile some very simple ladder logic programs, and print out some human readable version of the target code produced.

Weeks 11 and 12

Complete code for the interpreter/simulator, or the environment in which to run the generated Java or C target code.

Prepare further test cases. Review timetable for the remainder of the project and adjust in the light of experience thus far. Write the Progress Report drawing attention to the code already written, incorporating some examples, and recording any augmentations which at this stage seem reasonably likely to be incorporated.

Milestones: Simple ladder logic programs should now compile and run correctly, but probably with some serious inefficiencies in the code. Progress Report submitted and entire project reviewed both personally and with Overseers.

Weeks 13 to 19 (including Easter vacation)

Rework the entire implementation to enhance the richness of the language it can deal with. *<It is possible that the initial implementation has severe restrictions on the number of relays or coils that can be handled. Such restrictions can be freed at this stage.>* Write initial chapters of the Dissertation.

<The Easter break from lectures can provide a time to work on a substantial challenge such as the computation of logarithms, where an uninterrupted week can allow you to get to grips with a fairly complicated algorithm. This is a good time to put in some quiet work (while your Supervisor is busy on other things) writing the Preparation and Implementation chapters of the Dissertation. By this stage the form of the final implementation should be sufficiently clear that most of that chapter can be written, even if the code is incomplete. Describing clearly what the code will do can often be a way of sharpening your own understanding of how to implement it.>

Milestones: Preparation chapter of Dissertation complete, Implementation chapter at least half complete, code can perform a variety of interesting tasks and should be in a state that in the worst case it would satisfy the examiners with at most cosmetic adjustment.

Weeks 20 to 26

<Since your project is, by now, in fairly good shape there is a chance to use the immediate run-up to exams to attend to small rationalisations and to implement things that are useful but fairly straightforward. It is generally not a good idea to drop all project work over the revision season; if you do, the code will feel amazingly unfamiliar when you return to it. Equally, first priority has to go to the exams, so do not schedule anything too demanding on the project front here. The fact that the Implementation chapter of the Dissertation is in draft will mean that you should have a very clear view of the work that remains, and so can schedule it rationally.>

Work on the project will be kept ticking over during this period but undoubtedly the Easter Term lectures and examination revision will take priority.

Weeks 27 to 31

<Getting back to work after the examinations and May Week calls for discipline. Setting a timetable can help stiffen your resolve!>

Testing and evaluation. Finish off otherwise ragged parts of the code. Write the Introduction chapter and draft the Evaluation and Conclusions chapters of the Dissertation, complete the Implementation chapter.

Milestones: Examples and test cases run and results collected, Dissertation essentially complete, with large sections of it proof-read by Supervisor and possibly friends and/or Director of Studies.

Weeks 32 to 33

Finish Dissertation, preparing diagrams for insertion. Review whole project, check the Dissertation, and spend a final few days on whatever is in greatest need of attention.

<In many cases, once a Dissertation is complete (but not before) it will become clear where the biggest weakness in the entire work is. In some cases this will be that some feature of the code has not been completed or debugged, in other cases it will be that more sample output is needed to show the project's capabilities on larger test cases. In yet other cases it will be that the Dissertation is not as neatly laid out or well written as would be ideal. There is much to be said for reserving a small amount of time right at the end of the project (when your skills are most developed) to put in a short but intense burst of work to try to improve matters. Doing this when the Dissertation is already complete is good: you have a clearly limited amount of time to work, and if your efforts fail you still have something to hand in! If you succeed you may be able to replace that paragraph where you apologise for not getting feature X working into a brief note observing that you can indeed do X as well as all the other things you have talked about.>

Week 34

<Aim to submit the dissertation at least a week before the deadline. Be ready to check whether you will be needed for a viva voce *examination.>*

Milestone: Submission of Dissertation.