RTFM: A Digital Design Lab Expert

by

Owen B. Wessling

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degrees of Bachelor of Science in Computer Science and Engineering and Master of Engineering in Computer Science and Engineering at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1996

© Owen B. Wessling, MCMXCVI. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.

Author

Department of Electrical Engineering and Computer Science

December 18, 1995

Certified by

D. E. Troxel
Professor
Thesis Supervisor

Accepted by

F. R. Morgenthaler
Chairman, Departmental Committee on Graduate Students
RTFM: A Digital Design Lab Expert

by

Owen B. Wessling

Submitted to the Department of Electrical Engineering and Computer Science
on December 18, 1995, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Computer Science and Engineering

Abstract

This thesis describes the design and construction of the Resource Teaching Frame-based Machine (RTFM), a knowledge based (or expert) system which helps users to debug and design digital systems. The RTFM system seeks to act in a more educational manner rather than as a direct problem solver, as it is intended for primary use within introductory digital design courses. The project, while seeking to help users with general problems when asked, is primarily based upon problems and questions students are likely to encounter in the 6.111: Introductory Digital Design Laboratory class at the Massachusetts Institute of Technology.

Thesis Supervisor: D. E. Troxel
Title: Professor
Acknowledgements

I would like to first thank all the people — students, teaching and lab assistants, and professors — who helped me discern the knowledge that went into the RTFM system, for, without them, this thesis would not have been possible. Secondly, I would like to thank all the teaching assistants and students who will come after me and help to expand the RTFM system with their ideas and knowledge, for, without them, the RTFM system will never achieve its full potential.

I would also like to thank my parents for their continual support for all my endeavors, particularly through the last several years, and my friends who put up with me while this thesis supplanted\(^1\) my life.

\(^1\text{An apt term — according to the }\text{webster}\text{ program: “supplant (2a2): to eradicate and supply a substitute for”}.
Contents

1 Overview ........................................... 8

2 Purpose and Reasons, or How Can RTFM Help? ............ 10
   2.1 An Expert System Could Help .......................... 11
   2.2 The General Layout of the RTFM System ............... 13

3 The Knowledge Base for RTFM .......................... 14
   3.1 Matching the Representation with that of the Experts .... 15
   3.2 Frame-Based Knowledge Bases ......................... 17
   3.3 The RTFM Knowledge Base ............................ 19
      3.3.1 Implementation ................................ 21
      3.3.2 Gleaning Knowledge ............................ 29

4 The Supervisory Engine for RTFM ........................ 31
   4.1 The Parts of the Supervisory Engine .................. 31
      4.1.1 Frfun.l ....................................... 32
      4.1.2 Super.l ....................................... 34
      4.1.3 RTFMinit.l .................................... 37
   4.2 Operation of the Supervisory Engine .................... 37
      4.2.1 RTFM Search .................................. 39
      4.2.2 Multiple Answers .............................. 40

5 RTFM User’s Guide .................................. 42
   5.1 Overview .......................................... 43
5.2 Inside RTFM ........................................ 44
5.3 Example ........................................... 45
5.4 Bugs and Exhaustion ............................... 50

6 RTFM Programmer’s Guide ................. 51
   6.1 Welcome! ......................................... 52
   6.2 Ideology .......................................... 52
   6.3 What You Will Need ............................. 53
   6.4 What Should You Change? ....................... 53
   6.5 What is a Node ................................... 57
       6.5.1 A Few Tricks ................................. 58
       6.5.2 How to Create a Node ....................... 59
   6.6 Parting Words ................................... 59

7 Conclusions ........................................ 61

A Knowledge Base Code Listings............... 64
   A.1 Chips.l ........................................... 65
   A.2 Kit.l ............................................. 78
   A.3 Tools.l .......................................... 83
   A.4 System.l ......................................... 111
   A.5 Design.l .......................................... 113

B Supervisory Program Code Listings .......... 120
   B.1 Frfun.l .......................................... 121
   B.2 Super.l .......................................... 126
   B.3 RTFMinit.l ...................................... 128
# List of Figures

3-1  A section of the knowledge network, before and after an activation. . 17  
3-2  Analogous Pattern Matching with a Traditional Frame–Based System 19  
3-3  Analogous Pattern Matching with the RTFM System . . . . . . . . . 20  
3-4  Chip Sub–Knowledge Base Network . . . . . . . . . . . . . . . . . . 24  
3-5  Kit Sub–Knowledge Base Network . . . . . . . . . . . . . . . . . . 25  
3-6  Tools Sub–Knowledge Base Network . . . . . . . . . . . . . . . . 27  
3-7  System Sub–Knowledge Base Network . . . . . . . . . . . . . . . . 28  
3-8  Design Sub–Knowledge Base Network . . . . . . . . . . . . . . . . 29  

4-1  Information Flow in the 6.111 Lab Expert . . . . . . . . . . . . . . 38  
4-2  Overlapping Pattern Matching Concept . . . . . . . . . . . . . . 41  

6-1  Chip Sub–Knowledge Base Network . . . . . . . . . . . . . . . . 55  
6-2  Kit Sub–Knowledge Base Network . . . . . . . . . . . . . . . . . . 55  
6-3  Tools Sub–Knowledge Base Network . . . . . . . . . . . . . . . . . . 56  
6-4  System Sub–Knowledge Base Network . . . . . . . . . . . . . . . . . . 56  
6-5  Design Sub–Knowledge Base Network . . . . . . . . . . . . . . . . . . 57
List of Tables

3.1 Node Slot and Data Type Summary. .......................... 22
6.1 Node Slot and Data Type Summary. .......................... 57
Chapter 1

Overview

The RTFM program is an expert system which is designed to help users debug and design digital systems. It has been observed that many students beginning to explore the digital design process are in need of help beyond the basic concepts of digital design. In particular, the application of those concepts and the physical debugging of systems are often mystifying to the digital design initiate. Most introductory courses try to meet this need with assistants who staff the laboratories to help students with their projects; however, there are usually a few inherent problems with this solution — problems which the RTFM system seeks to address.

This educational purpose is important to the design of the RTFM system. The RTFM program is designed to present information in a manner which should help students to think about how their system is functioning rather than providing a complete solution to a problem. In this manner, RTFM is intended for use as an educational tool, suitable for classes about digital design, like 6.111: Introductory Digital Systems Laboratory at the Massachusetts Institute of Technology. The goal of RTFM is not to solve students’ problems, but to help them to solve their problems.

To accomplish this goal, the RTFM program uses a network of knowledge nodes. These nodes are activated as the user answers questions posed by the RTFM program. Upon activation, a node can present some information to the user and/or ask further questions about the system being described by the user. As the process continues, the RTFM program narrows the area of the network, and the number of applicable
nodes, related to the user's problem; as RTFM closes in on the problem, information is presented to the user that ought to help him figure out the problem. Eventually, the problem is identified and specific information relating to the solution is given to the user.

This document endeavors to describe the RTFM system in its entirety, beginning with the purpose and reasoning for the expert system and then continuing with the design and development of the knowledge base which is central to the RTFM system. We then discuss the supervisory program which allows users to traverse the knowledge base. Two guides to the RTFM system are then examined: one for the users of the system; and one for those wishing to extend the RTFM program, itself. Finally the RTFM system as a whole is examined to conclude how well it fulfills its goals. In the appendices you will find source code listings for the RTFM program.
Chapter 2

Purpose and Reasons, or How Can RTFM Help?

Debugging lab projects is one of the most time-consuming and initially mystifying aspects of the digital design process, particularly for people new to the discipline. Most students taking a digital design class enter it with little or no design experience and even less actual hands-on experience. The consequence of this is that many students spend a large amount of time in lab, particularly during the beginning of the course, wondering what is wrong with their project and trying to debug the project using poor technique. Teaching assistants are often asked to staff the lab because of this, and they do provide a great deal of help in such cases, but there is room for improvement because of the following situations:

- While lab assistants do provide a great deal of assistance, they are not always around. Quite simply, the teaching assistants, inevitably, cannot spend all their time in lab. Particularly for those students working late, the benefit of a teaching assistant is often unavailable.

- Many problems are trivial; while these problems are just as important to the students they affect, the lab assistants’ time could be better spent dealing with more complex problems. Lots of problems of various sorts affect the students. However, by and large, the more common problems are fairly easy to solve —
perhaps a control pin is asserted incorrectly; maybe the student forgot to wire power and ground. These problems take only a moment of the lab assistant’s time, but if he needs to deal with a lot of them, he may not be able to spend the time necessary on the more complicated problems that do occasionally occur.

- Lab assistant turnover is fairly high. Each lab assistant is working only for about a year, after which they leave and take all their knowledge with them. Very few leave any documents or non-verbal legacy of the knowledge they have gained.

- While each new lab assistant must have a good understanding of digital design and the debugging process before becoming an assistant, there is still a considerable amount that a new assistant has not encountered. Furthermore, there is no current record of commonly encountered problems. These combine to produce a considerable rampup time for new lab assistants to prove helpful in a large variety of situations.

While none of these problems seriously threaten the ability of the students to learn digital design skills, they do cause some misgivings. Students often despair when they cannot figure out what is wrong with their project and no one is around who can help them. Almost as bad, in terms of helping the student understand the material, is when a student has a problem and a lab assistant looks briefly at the project, moves a wire or two, and fixes the problem. Often the assistant does not have the time to explain what was wrong or how he knew it was wrong, and while this fixes the instance of the problem, the student is still left mystified. Consequently, a system which could help to alleviate some or all of these problems could benefit the students and staff of a digital design course.

2.1 An Expert System Could Help

The nature of the present problems match very closely to the sort of problems that Davis describes as being well suited to a knowledge based (expert) system solution
[9]. In general the problems revolve around the possession and use of knowledge:

- The lab assistants are valuable and needed because they possess knowledge about debugging lab projects that students lack. This seems obviously true — if the teaching assistants were not helpful in trying to aid students, their duties would not include staffing the lab.

- Knowledge is always being lost because lab assistants are constantly leaving. Noting that new lab assistants learn a little more with each problem they encounter, we must admit that they have gained knowledge by the time they leave. However, little of this knowledge is carried over from year to year as assistants are constantly leaving and being replaced.

- New lab assistants are not as familiar with all the possible problems because they have not encountered everything the older lab assistants have. This is something of a corollary to the previous point, since little of the knowledge of older assistants is passed on, new assistants must re-experience a number of problems before possessing a similar amount of knowledge as the departing assistants.

An expert system can address these problems quite effectively. Primarily, of course, the expert system is available as long as the user has access to a computer that provides it. As most digital design labs (and particularly the 6.111 lab) use computers to help in the digital design process, this should translate into the expert system always being available. Secondly, an expert system, assuming it is updated periodically, acts as a knowledge repository. As old assistants leave, significantly less of their knowledge leaves with them. Also, new assistants may use the expert system’s knowledge base to “jump start” themselves, decreasing the amount of time and number of experiences they require to achieve a higher level of knowledge.

Now, one minor point is that if the expert system requires updating, then it clearly will not be able to answer every question that the user might have. While this is true, the system will be quite capable of handling the trivial cases which occupy so much
of the assistants’ time. Thus, an expert system will not, nor should it, completely replace the teaching assistants; it should provide sufficient knowledge and help that those assistants can apply themselves more effectively to the complicated problems that arise.

Thus, an expert system that could help people to learn to design and, more importantly, to debug their digital systems would be of great benefit to students and staff. Such a system would alleviate some of the frustration experienced by students when they fail to determine why their digital project is not working on their own. That same system would provide staff with a means to maintain knowledge gained by teaching assistants as well as help new assistants to more quickly gain a high level of digital experience. It is with these goals in mind that the RTFM expert system was created.

### 2.2 The General Layout of the RTFM System

The RTFM expert system is separated into two primary parts: the knowledge base, which is the heart of the expert system; and a supervisory program which allows users to traverse the knowledge base and find the information which will help them. The knowledge base is the center of the RTFM system as this is where the knowledge for helping people to design and debug digital projects is kept. However, the knowledge base is insufficient by itself as it is just that — a collection of knowledge; as such we need to create a second subsystem, a supervisory engine, which can direct users through the knowledge base such that they find helpful information. Then, the combination of these two subsystems will provide a valuable service to the students and staff of an introductory digital design course.
Chapter 3

The Knowledge Base for RTFM

The knowledge base for the Resource Teaching Frame–based Machine (RTFM) has been constructed by following a particular sequence of investigation concerning the knowledge to be duplicated. That investigation began with looking for the “experts” in the digital design process; then finding the information they have; and then discerning how the experts use and apply their knowledge. In order to build a useful knowledge–based system, it is necessary to find out what knowledge is used by the human experts and how they use it. This is no simple task; finding the information itself is often not too difficult, but determining how they use it can often be quite difficult — quite often an expert has an intuitive feel about what they do and so some effort is required to articulate how they combine and utilize information to solve a problem.

We find that the knowledge base must be an effective storage of both the information that experts have, and the methods by which they use that information. A system that has access to only one of these — either the information or the process — will inevitably be inferior to a system that can successfully replicate both an expert’s information and the process by which it was accessed. In order to effectively combine these two primary requirements of the knowledge base, it is desirable to find a method of knowledge representation that complements the experts’ process. Thus, when we access information, we can fairly easily replicate the expert’s process of using that information, as well as to apply it to the problem at hand.
3.1 Matching the Representation with that of the Experts

The teaching assistants (TAs) which made up the bulk of the experts used to develop the RTFM system showed some particular traits in how they used information to help students design and debug projects. First and foremost was that the TAs carried on a running commentary of the problems and/or gave a very large amount of information that encompassed a number of possible causes that the student should think about. That is, if the TA had time to actually sit down with the student, the debugging process would look something like\(^1\):

Student: The problem looks like <foo>.

TA: Well, what might cause that?

Student: I don’t really know.

TA: OK, well, think about all the inputs. How do they control this chip?

Student: Well, the inputs do ...<description>.....

TA: OK, now consider the <foo> behavior, how could an input generate this?

Student: Well, input one might do that if it was tied low.

TA: Anything else?

Students: Well, input two might do that if it were glitchy.

TA: OK, plug those into the logic analyzer and see what’s happening.

This discussion aptly shows the “running commentary” effect of the expert that we seek to duplicate in the RTFM system.

Given that the TA had a number of people waiting for help, that same conversation might occur as:

\(^1\)These are fictitious dialogues. They do not demonstrate an actual conversation, but do exemplify the type of interaction between an expert and a user.
Student: The problem looks like <foo>.

TA: Well, that chip should act like <this>. Acting like <foo> is not unreasonable if some of the control signals are incorrect. Particularly inputs one and two could cause that.

Student: Um, ok.

TA: Wire those up to the logic analyzer and see what’s happening, you ought to be able to see what’s going on then. Give a yell if neither of those inputs looks to be problematic.

This discussion illustrates the hurried assistant’s tendency to select the alternate method of forcing student deduction — give them a lot of applicable information and tell them to explore it and determine what applies to the current problem.

The point of these dialogues is that an expert uses an ongoing information trickle and tries to avoid giving a direct solution. The first example details a narrow information bandwidth (lots of questions) where as the second shows a much wider bandwidth (there is a much higher information-to-question ration), but in both cases we see that the expert does not perform a comprehensive evaluation of the system and then spit out a definite solution. Instead, the expert gives some information as the problem develops, directing how the search ought to proceed and then, given that a particular detail could be the problem, directs the student to investigate that. If that detail is the problem, great; the student is done and can continue; however, if not, there is no big problem; the TA gives a little more information on where else the problem could come from and the search proceeds. Thus, information is given either a little at a time, or in huge chunks, trying to push the student’s own deductive abilities as far as possible.

This is the information process behavior which I deem critical to the RTFM system. I feel it is necessary that RTFM not work in a manner that asks a large number of questions and then “magically” tells the user what is wrong. Far more important, it seems, is the process of giving information continuously and detailing how the search should proceed. A large part of learning to debug is not in the solution, but in the
process; and so if RTFM is to be successful, it needs to duplicate this demonstration of process. Users need to be shown how to investigate digital problems, not just have the problems solved.

### 3.2 Frame–Based Knowledge Bases

To accomplish this, I chose to implement the RTFM knowledge base in a frame–like representation network. The traditional frame method makes use of a few large information nodes which contain the majority of the knowledge given to the user and a network of question nodes to determine which frame(s) of knowledge to present to the user. RTFM works in a similar manner, but tries to distribute the information a little more universally.

This modified style of knowledge representation uses a network of interconnected nodes to hold and access the available information (see Figure 3-1). RTFM works by activating sections of the network based on answers to questions provided by the
user. Each node exists in one of two states: active or inactive. Initially, nodes are inactive, though a startup node is activated upon starting the RTFM system. As seen in Figure 3-1, a node may branch to several other nodes, each of which could be activated depending on the information given by the user; in the case shown, only one new node is activated when the user responds affirmatively that he is counting with the LS163 chip.

Each link between nodes represents a means of activation. When one node is activated, it may direct the system to ask a question of the user and/or present some useful information to the user. Depending on the user’s response, other nodes may be activated. As the user enters information about his problem, the system follows a cyclic pattern of: query → response → activate → query.

Traditionally, as with the system described by Palker [8], nodes can be of different types, some being query-nodes and some being frame-nodes, or frames. The frames represent a known piece of information such as a particular problem or disease or whatever; each of which have various conditions of activation. The network of query-nodes acts in a manner such that more query-nodes are activated appropriately to try and satisfy condition statements for the frames. These conditions are kept around in a simple memory buffer and as a frame’s conditions are met, the information it contains is given to the user. Usually there are substantial conditional requirements for a frame and so only a single frame can be matched precisely. Given a case where no frame is exactly matched, the system usually responds with a list of the frames which most closely match the user’s description.

In this manner, the frame-based representation can be thought of as a means of pattern matching. The user outlines a particular problem (by responding to the proposed queries of the expert system) and the system tries to match the described problem to those that it has as “example” cases (see Figure 3-2).
3.3 The RTFM Knowledge Base

This is where RTFM differs slightly from the traditional frame-based system. RTFM does not differentiate between frame-nodes and query-nodes. Instead, RTFM uses something of a hybrid which allows both information content and a query in each node. The effect of this is to allow RTFM to have a "running commentary" effect like the one used by teaching assistants. Instead of lumping all the information of a problem into one large node that is activated at the end, RTFM distributes that information through the querying process and so gives users the opportunity to see how a question is applicable to the situation. In the more traditional system, a user might answer several direct questions and then see an end description like:

Your system's response of <foo> to <test 3> indicates that you might have tried to use an improper family/pinout code while burning your PAL.

This information is in the past tense and more difficult for the student to assimilate into their thinking process because they are not actively looking at that part of the investigation. However, RTFM could instead give the same information in the process of helping the student:

Using an improper family/pinout code can often cause problems. Try performing <test 3> on your system. What is the response:
<foo>  
<bar>  
<baz>

```
When the user selects <foo> then RTFM may indicate that, indeed, it is likely that the family/pinout code was the problem. While this may seem like a minor difference in behavior between RTFM and the traditional frame–based systems, it is also central to the ability of RTFM to match the process by which TAs help students. In essence, the RTFM system is allowed more freedom to make suggestions about what the problem might be. In many traditional frame–based systems, the use was of a nature in which the diagnosis really needed to be right the first time. However, for RTFM, there is not a problem if a sub-module is modified and it turns out to not be the problem — the student spends a little extra time in lab (for which he probably ends up with a slightly better understanding of his system) and goes on to try another possible problem.

This difference also provides a small modification in the philosophy behind the pattern matching. In the traditional system, the match comparison was made for a single large chunk of information. Within RTFM, the matching behavior acts more like what is described in Figure 3-3. Instead of matching a single large chunk, RTFM matches small pieces of information, kind of like piecing together a jig–saw puzzle. Indeed, this concept is a direct correlation of the idea of showing the TAs' process — we want to show students how to diagnose a problem, and how to go about making that diagnosis, not just tell them what the problem is. We want to show them the pieces that make up their problem, not just the problem itself.

Thus, RTFM's network consists of only a single type of node which can contain
both information and questions, though the nodes can be separated into three groups: those which contain information, but no question; those that contain a question, but no information; and those that contain both information and a question\(^2\). Upon activation the node will present its information to the user and add its query to the query stack of the RTFM supervisory system.

The RTFM system also presents information in a wide-bandwidth fashion; that is, if a user responds to a question, RTFM will tend to respond with information rather than more questions. The reasoning behind this is that once a user has identified the major block which is causing a problem, RTFM (like the TAs) could either try and narrow the problem exactly, or give the student a lot of information and tell them to figure out what part of it applies to them. This latter method seems better for a number of reasons. First, it forces the user to think a little more about what is going on. Secondly, the user is more likely to use the program as he will not feel that most of his time is being spent answering meticulous questions. Finally, it is easier on the programmer as the knowledge network becomes much less complicated. Thus, a wide-bandwidth approach seems like a good idea all around.

### 3.3.1 Implementation

The RTFM knowledge base is implemented in Common LISP [4], using the FROBS platform presented by Muehle [7]. FROBS is a library which provides procedures and functions to simplify the construction of a frame-based knowledge system in Common LISP. Initially, the Kyoto Common LISP [11] variation was used; however, to improve portability the RTFM system was modified so as to utilize only Common LISP standard functions. In this manner, any flavour of Common LISP derived environments may be used with the RTFM system.

A number of procedures have been developed to provide a simple method for creating nodes in the knowledge base. The Make-node function provides an easy means of creating nodes for the RTFM knowledge base; the syntax for this procedure

\(^2\)These groupings, however, are often somewhat misleading — it is not uncommon for the wording of a question to present as much, or more, information as a node that has only “information”.

21
Table 3.1: Node Slot and Data Type Summary.

<table>
<thead>
<tr>
<th>Slot</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>Symbol</td>
</tr>
<tr>
<td>Info</td>
<td>List of Strings</td>
</tr>
<tr>
<td>Query</td>
<td>List of Strings</td>
</tr>
<tr>
<td>Links</td>
<td>List of Response-Nodes Lists</td>
</tr>
</tbody>
</table>

is simply:

```lisp
(make-node id info query links)
```

Where id is the name of the node, given as a symbol. Info and query are lists of strings, which provide the node’s information content and question, respectively. And links is a list of sublists matching possible answers to the nodes they should activate (these slot datatypes are summarized in Table 3.1). For example:

```lisp
(make-node 'nodePALASGNproblem nil

'("Which of the following error messages most closely matches"
"your message or problem:"
"Trying [H]ueristic 3..."
"Warning: foo cannot be [a]ssigned as specified"
"Cannot allocate [p]in for <bartype> output foo, abort"
)
'((h {nodePALASGNheuristic3})
(a {nodePALASGNassign})
(p {nodePALASGNallocate})
)
)
```

Here we see a node for which the id (giving the name of the node) is “nodePALASGNproblem”. There is no immediate info for this node (that slot is nil). The query is the list of strings beginning with “Which of the following...” and the links of the node recognize the responses of “h”, “a”, and “p”. Note that for any of the three list parameters, if the user does not wish to enter anything, a nil entry is appropriate. For more information about making nodes and working with the knowledge base refer to the RTFM Programmer’s Guide in Chapter 6.
This simple-to-use node creator was developed in the hope that should the RTFM system come into common use by a class that it will not be considered a complete black box. That is, it is hoped that the assistants of any such class will feel comfortable changing, modifying, and updating the RTFM system as more problems are identified and solved and new or different equipment is placed in the labs. In this manner, the RTFM system will continue to preserve knowledge from a transient expert group. As well, RTFM can provide a means for new TAs to look at the history of some of the simpler problems commonly encountered in their class, thus giving a better initial understanding of the problems they are likely to encounter.

The knowledge base is separated into five primary areas of expertise: common chips; the lab kit; lab tools; system information; and design knowledge. We will now look at each section of the knowledge base in more detail.

Chip Knowledge

This section of the RTFM knowledge base deals with the chips commonly encountered in digital design, with particular emphasis on the chips used in the 6.111 course. As chips are the basic building block of digital systems, it is important to understand what a particular kind of chip is and how to interface with it. The majority of this information is not properly in the scope of the RTFM system, in that the designer should have read the detailed specification sheets for any chip he is using; however, many common mistakes are made from misinterpreting an interface or a presumed function. Thus, some of this information is in the scope of RTFM and so some knowledge about common chips used in digital design is necessary.

In particular, the chip portion of the RTFM knowledge base seeks to give a user information about any particular chip that could relate to problems he is experiencing and which he has traced to that sort of chip. This information includes the chip's pinout, as often a mistake is simply because a pin is misconnected, or a signal is believed to be its inverse, and so the pinout can help a user to verify the proper

---

3 As of this writing, the chips section contained information on about a third of the 'LS TTL chips and about a half dozen other chips including an SRAM, PALs, UART, and a microsequencer.
configuration. As well, any problems which tend to occur with a particular chip, are noted as well. For example, the information for the LS163 is:

\[ \begin{align*} +-----------------------------+ \\
\text{The LS163 is a synchronous 4-bit counter with SYNCHRONOUS} & \text{ Clear and Load.} \\
\text{/CLR} & 1 \quad \text{16}/Vcc \\
\text{CLK} & 2 \quad \text{15}/RCO \\
\text{A} & 3 \quad \text{S} \quad 14/Qa \\
\text{B} & 4 \quad \text{13}/Qb \\
\text{C} & 5 \quad \text{6}/Qc \\
\text{D} & 6 \quad \text{3}/Qd \\
\text{ENP} & 7 \quad \text{10}/ENT \\
\text{GND} & 8 \quad \text{9}/LOAD \\
\end{align*} \]

If you are having problems counting at all, check that your \$/CLR and \$/LOAD signals are correct. Also, both \$ENT and \$ENP must be high to enable counting.

If counting is occurring, but not at a proper pace, you may have a problem with your reset rates. Also, if you are cascading multiple counters, you should be careful how you enable your '163 --- in that RCO is only valid while \$ENT is high, thus you will have problems counting with multiple chips if you hold \$ENT high and pulse \$ENP. If your counting pattern is off by exactly one, you may want to check your reset circuit, or, if the reset is RCO triggered, check that your \$ENT control allows for proper triggering."

The information contains a brief description of what the chip is as well as a pinout diagram of the chip — the LS163 is a 4-bit synchronous counter; it is housed in a 16-pin package with the interface shown above. Also, some of the more common errors made with the '163 are discussed: misuse of the control signals which disable counting, as well as a warning regarding the cascading of several '163s in succession (which is a common problem in 6.111).

The chip superknowledge base of the RTFM system, as with many of the sub-bases,
is fairly self-contained with only some crossover into other parts of the knowledge base. Those crossovers occur principally with some of the more complicated and programmable chips for which a user might request information about the chip, but really want information about a problem with programming that chip. A diagram depicting the general chip network is shown in Figure 3-4. The code for the chip sub-base is listed under chips.1 in Appendix A.

**Kit Knowledge**

The development kit, which provides students taking 6.111 with a portable digital design environment, has a few problems which are directly related to it. As such, occasionally a student has a problem which is either clearly a kit problem (such as a burned out LED) or perhaps is based in the kit, though the symptoms lie elsewhere (such as with many power problems). To handle these situations, RTFM needs to have some knowledge about these kit problems, how to recognize them, and how to deal with them.

A diagram outlining this sub-knowledge base is shown in Figure 3-5; as well, code for this section is also in Appendix A, under kit.1.
This sub-knowledge base provides information on topics ranging from basic power problems to the kits’ primary input devices (switches) to the several output devices of the kit (LEDs, Hex LEDs, and the Logic Analyzer interface).

In general, this section of the RTFM knowledge base helps students to determine if a problem is caused by the kit, and if so directs them to take their kit to the equipment desk for fixing. Unlike most other sections, this part of the knowledge base tries to help a user pinpoint exactly where a problem is (if it is in the kit) and then either gives specific instructions for fixing it (usually telling them to get it repaired at the desk). This is because the kit is not intended to be an obstacle in the process of learning digital design, but a tool to help the student; if there is a problem with that tool, we want to identify it and fix it as soon as possible.

This sub-knowledge base also displays the multi-branching possibilities of the RTFM knowledge base. Examine the initial node in this sub-base:

```
(make-node 'nodekitproblems
 nil
 '("Okay, so you think something is wrong with your kit. First off, have you"
   "checked your kit to verify that the major components are working? Namely"
   "when you first got your kit, did you check to make sure all LEDs and switches"
   "worked and that power and ground were steady levels at +5, +12, -12, and 0"
   "Volts at the appropriate places? [y/n]"
   )
 '((y {nodekitbranch} {nodekitinfo})
   (n {nodekittest} {nodekitbranch} {nodekitinfo})
   )
   )
```

Notice that the links generated for each possible answer active more than one node, thus causing RTFM to have to explore multiple branches of information; more will be said about this later in the supervisory program section.

**Tools Knowledge**

Of critical importance to the success of learning digital design is the understanding of how to use the tools available. These tools make the design of large, complex systems much more manageable. 6.111, and just about any other digital design class or lab, has a number of tools to help do this. Success in designing a system often requires
the successful use of some or all of these tools; but, as these tools are usually black boxes, when an error occurs it may leave the user wondering exactly what has gone wrong. RTFM attempts to help students by dedicating part of its knowledge base to helping students figure out what is going on with these programs and devices.

Most of the programs used in 6.111 provide fairly good error detection and give reasonable error messages to students when a problem arises. However, sometimes they do not and sometimes the error is of a kind that simply eludes the perception of the student. RTFM seeks to help students when these problems arise by giving a full interpretation of the various error messages of these programs and explaining the various errors that can generate the message the student received.

A diagram showing the tools sub–knowledge network is in Figure 3-6; and the code is in Appendix A, under tools.1.
System Knowledge

The final subsection of the RTFM knowledge base dedicated towards helping students fix problems is the systems sub-knowledge base. This sub-base acts as a buffer between the trivial problem information provided by the previous sub-knowledge bases and the highly complex systems that are beyond the abilities of the RTFM system.

The previous sub-knowledge bases look mostly at a single element of a system. This is a good thing, as most problems can be traced back to a single chip, or line of code, or kit element; however, it is often the case that a student does not know how to find it. One of the primary premises for the need of this system is that most students who are beginning to learn digital design do not know how to go about debugging a complex system. It is to this problem that the system sub-knowledge base is directed.

Most complex systems are built under particular paradigms; for 6.111 the two common paradigms are the finite state machine (FSM) and the microprogrammed controller unit (MCU). The system sub-base seeks to provide information which helps students to figure out how to debug these kinds of designs in general, because both of these paradigms can be debugged in a vary straightforward process. In addition to these two specific paradigms, the system sub-base provides a supergeneric debugging node which gives a very general process for debugging any digital system.

Figure 3-7 provides a diagram showing this sub-base's network; and again, the code for the system sub-knowledge base is in Appendix A, under system.1.

---

4if the process is known...
Design Knowledge

The design sub-knowledge base is an attempt at a preemptive strike against system bugs. This part of the RTFM knowledge base seeks to provide students with some additional information and a little help for thinking about the projects presented in 6.111. This is a very unusual section in that it seeks to show students problems they might run into and to give them ideas about how to design a project without actually giving them any of the design.

This sub-knowledge base is divided into categories based upon the project students may wish to consider. Various lab projects and submodules are discussed in this sub-base as well as some topics that are not directly connected with a particular lab, but may crop up in any project (like bus contention).

A diagram depicting the network of this sub-base is shown in Figure 3-8; the code for this sub-base is under design.1 in Appendix A.

3.3.2 Gleaning Knowledge

The information contained within the RTFM knowledge base was gathered together by several means so as to cover as much material as possible. These methods of acquisition included recording knowledge from myself and other teaching assistants, experimentation with tools and chips, and researching information in lab manuals. The combination of these activities provides a reasonably large base upon which to begin the RTFM knowledge base. Certainly, the knowledge base is not complete,
nor will it ever be, as there will always be some knowledge that it does not contain; however, the sources mentioned above provide a good initial content for the system.
Chapter 4

The Supervisory Engine for RTFM

In addition to the knowledge base, the RTFM system requires some sort of program that allows users to traverse that knowledge base and extract the information they need. This aspect of RTFM is provided by several external procedures and objects that are collectively referred to as the supervisory engine of the RTFM system. The code that describes these various procedures and objects is located in Appendix B, under the headings Frfun.1, Super.1, and RTFMinit.1; each heading corresponds to a separate file in the Common Lisp implementation of the supervisory engine. We will first examine each of these files to see how the engine is put together, and then we will continue to discuss the operation of the engine in general.

4.1 The Parts of the Supervisory Engine

The supervisory engine of the RTFM system is separated into three files, each of which contains procedures and objects slightly different in scope than the other two. Frfun.1 contains the loose procedures which provide low-level functionality of the engine as well as abstraction barriers between the rest of the RTFM system and the current implementation. Super.1 contains the primary supervisory procedure which repeatedly directs questions to, and answers from, the user and adjusts the current
state of the knowledge base accordingly. RTMinit.1 contains a simple startup routine for the KCL platform to easily load the FROBS [7] package and the RTFM system into memory.

### 4.1.1 Frfun.1

Frfun.1 is a collection of loose procedures and object definitions that provide the low-level functionality and high-level abstractions of the RTFM system. The code for the this file is in Appendix B.

There are two objects defined for the RTFM system in Frfun.1: the memory object and the node object. The memory object is used to keep track of the state of the knowledge base with the aid of its three slots: activated, queries, and links. The activated slot simply keeps track of what nodes in the knowledge base have been activated; the means of keeping this current will be discussed momentarily. The queries and links slots maintain lists of the query and link pairs of the activated nodes. In this manner, the memory node acts as a short-term memory device which keeps track of what questions need to still be asked of the user and what to do upon the responses given by the user. To implement this “short-term memory”, an instance of the memory object called mem is created for the supervisory engine.

The node object is the basic building block of the knowledge base. It is defined, as was previously described, with three information slots: an info slot for holding general information; a query slot for holding a question; and a links slot for holding a list of response—nodes-to-be-activated sublists. All of these slots are lists of either strings (info and query slots) or smaller lists (links slot). In addition to these three primary slots, two other slots are defined as well: an id slot is defined which gives a means to identify a particular node; and a state slot which indicates whether a node is active or inactive. The three information slots are assumed to be nil and the state slot is assumed to be inactive when a new instance is created.

Frfun.1 implements a large part of the bookkeeping of the supervisory engine by defining a daemon over the state slot of the node object. This daemon, and each instance of the node object has one, is instructed to perform certain tasks if the
The `state` slot is set to `active`. Should this occur, the daemon is instructed to print any information in the `info` slot immediately, and, assuming `query` is not `nil`, push the `query` and `links` information onto the stacks in `mem`. Also, the newly activated node’s `id` is added to the `activated` slot in `mem` to let us keep track of what is going on. Finally, since all the information from the node has been used, the daemon resets the node’s state to be `inactive`\(^1\). This greatly simplifies the behavior of the rest of the supervisory engine as we always know what nodes have been activated, and we know that all `info` information is handled automatically. The rest of the supervisory engine (contained in `super.1`) need only worry about handling the querying of the user and the appropriate activation of the knowledge base.

A number of procedures in `frfun.1` simply implement abstraction barriers to increase modularity and allow for easier changes to parts of the system that might be reworked. The first such procedure is the `make-node` procedure which greatly simplifies the coding of new nodes for the knowledge base. Recall that one of the desirable benefits of the RTFM system is to act as a repository for digital knowledge; well, if expanding or changing the knowledge base required a great deal of effort to figure out then it would not be likely to happen. As such, I have attempted to ease the matter by providing this very simple, very intuitive procedure to create nodes\(^2\). Also, the two functions `prt` and `query-user` are defined to allow the supervisory engine to abstractly print strings to the screen and to obtain a response from the user. Currently, these procedures provide a simple text interface between RTFM and the user; however, a more advanced GUI could be developed simply by modifying these two procedures (and perhaps adding a few lines in the initialization routine `RTFMinit.1`).

`ResetRTFM` is a procedure which allows for a modularized reset routine, which will place the RTFM system into its startup configuration. Also, the function `revexplode` abstracts some of the processing of the user-entered responses; this particular func-

---

\(^1\)This last step is done because it is not inconceivable that multiple branches will activate a single node from different directions; should this happen we want the node to re-present its information to the user, and for this to happen we must reset the node.

\(^2\)Indeed, if it is so desired, this also allows for someone to create an entirely different knowledge base on another topic entirely and still be able to reuse the supervisory engine of RTFM.
tion turns the response, which may be coded so as to indicate multiple answers, into a list of separate responses.

RTFM is designed such that in addition to normal question answers, a user may also enter a system command at the prompt. When this happens, the command is to be processed and the current question re-asked. The procedure \texttt{commandp} is called to test an argument to see if it is a system command, and if so processes that command, as well.

The remaining procedures in \texttt{frfun.1} are support procedures for \texttt{commandp}. That is, the functions \texttt{helpcommand}, \texttt{quitcommand}, \texttt{backupcommand}, and \texttt{activatecommand} all implement the respective system function when called by the \texttt{commandp} procedure.

Thus, \texttt{frfun.1} provides a number of procedures and objects that define the basis for the supervisory engine (and actually the knowledge base, too) and abstract many of those procedures so that improvements to the system can take advantage of a more modular design.

### 4.1.2 Super.1

The file \texttt{Super.1} contains the main procedure of the supervisory engine. The procedure in question is the function \texttt{expert} and it consists of initializing the RTFM system and then repeatedly querying the user and processing the user’s response. The processing consists of analyzing the response to determine what sort of action needs to be taken: an invalid response can be ignored; a system command needs to be processed; and a valid response needs to activate nodes. Eventually, the knowledge base becomes depleted and the list of queries is emptied, at which point \texttt{expert} indicates that the knowledge base has been exhausted and asks if the user wishes to quit using RTFM. A positive answer breaks the user out of the \texttt{expert} main loop and ends the program; a negative answer restarts the loop and RTFM is reinitialized for a new session. This loop works as simply as it does because of the \texttt{daemon} linked to each node — as soon as a new node is activated, it will automatically push its \texttt{query} and \texttt{links} onto the memory stacks. Thus, the \texttt{expert} procedure needs merely to repetitively pop those stacks, ask the questions of the user, and process the responses.
A more detailed description is this: **expert** is a function which consists of two loops, one nested within the other. The first loop is begun immediately and its activity follows the pattern:

1. Initialize the RTFM system. First, clear the memory of all queries and links, then activate the startup node. As previously mentioned, this should place a query—links pair into the memory stacks and so the main looping behavior may begin.

2. Now run the second, internal loop. More about this loop will be said in just a moment, but basically, we stay here until the knowledge base has been exhausted along the paths indicated by the user’s responses.

3. With the knowledge base exhausted, RTFM really can not help anymore, so we should refer the user and his problem to the teaching assistants who can better handle a new or complicated problem.

4. Finally, we should ask the user if they wish to exit the RTFM program. If so, then we can break out of this loop and end the program, otherwise we should start the loop again.

Now, we see that the second, internal, loop is where the program spends most of its time — in the repetitive cycle of asking users questions, comparing their answers to the question’s links, and then activating those linked nodes. A few special situations can occur here as well, which are:

- **Empty Memory Stacks**: If the *queries* slot of the memory object is ever empty at the start of this loop, then we cannot ask a question (there is no question to ask). This will only occur when the knowledge base has been exhausted, so when this occurs we break out of the internal loop.

- **An unrecognized answer is given**: If the user enters an answer at the prompt which the system cannot identify, then the RTFM system ignores that response and re-asks the current question.
A system command: If, during the previous situation, the entered answer is recognized as a system command, that command is processed before the question is reiterated. Available system commands will be discussed later.

Aside from these special cases, the internal loop asks the user a question and records the answer. It then compares that answer to the list of valid answers in the associated links list and picks out the list of nodes to be activated. It then proceeds to activate those nodes, whose daemons update the memory stacks and print out any available information.

System Commands

As previously mentioned, in addition to recognizable answers to questions, the user may also enter special system commands to the RTFM program. These commands provide a variety of functions to the user. To keep system commands from being likely answers to a normal question, all system commands are prefixed with the ‘!’ character; although the !help command, which can also be accessed by the character ’?’ is an exception to this. A list of system commands, their functions, and the triggering command follows:

- Help ![help or ?]: This command produces a brief message for the user which describes how to use the RTFM system and what commands are available in the system.

- Activate ![activate_{node}]: This command allows you to instantly activate any node in the knowledge base. Note that this command takes an argument — the command is !activate; however, in order to determine what to activate, the name of a node is needed. Furthermore, to ease parsing, the format is as a single word with a ‘_’ separating the command and argument. Thus, if a user wished to activate the node nodeLS151 they could do so by entering the command: !activate_nodeLS151.

- Quit ![quit]: This system command allows a user to terminate the current RTFM session. The system will ask the user if they want to completely quit.
the RTFM system; if not then the system is re-initialized for a new session, if so, then the RTFM program is terminated.

- **Backup ![backup]:** Occasionally, a user will find that the path through the knowledge base that RTFM is following is no longer the path he wishes to pursue. When this happens, instead of being required to either see the path to completion or quit and restart, the ![backup] command may be used to cease the current line of questioning and to move on to the most recent offshoot. After seeing the ![backup] command, RTFM will immediately remove the current node from the stack and continue the search with the most recent activated, but unprocessed node.

### 4.1.3 RTFMinit.l

The RTFMinit.l file contains the initialization routine for the RTFM system. By instructing a Common LISP editor to load this and only this file, the entire RTFM system can be initialized when RTFM is begun.

The file contains a number of load statements which bring the FROBS [7] package and then the RTFM system into memory. Then, a single startup node is created which can contain an introductory message to the users in its info slot, and of course, an initial query and links pair. And then we are ready to begin; the RTFM system is in memory, so the command (expert) initiates an RTFM session. When the user returns from the (expert) function call, they are done using RTFM, so we exit the Common Lisp environment.

### 4.2 Operation of the Supervisory Engine

The operation of the supervisory engine essentially provides a controlled search of the RTFM knowledge base and activates nodes as directed by the responses of the user. The overall activity follows a simple pattern of:

1. Provide any new (just activated) information to the user.
2. Ask the next question on the stack.

3. Retrieve the user's response.

4. Activate further parts of the knowledge base.

5. Begin the cycle again.

A diagram of the information flow is shown in Figure 4-1. The determination of how the knowledge base is searched is made by how, when new nodes are activated, their questions and links are placed into the memory stacks. This can be done consistently in one of two ways: either the new elements are stuffed into the bottom of the memory list; or they are pushed onto the top.

The first method would generate an effective queue of questions and links. Considering this, with a queue's FIFO behavior, the RTFM system would search through its knowledge network in a breadth-first manner. Each node that had been activated at a particular level away from the root would be examined before proceeding to the next level.
The other method, acting in a converse way, would behave like a stack. This time, the LIFO behavior of the stack would cause a depth-first search pattern where one path of the knowledge network would be examined until completed and then any branching paths would be followed up.

4.2.1 RTFM Search

While either of the above methods would work; the RTFM system is implemented using the stack/depth-first method. This method was chosen because the experts all seemed to perform in a depth-first manner when debugging a kit. If a problem seemed to be related to a particular chip or design, that aspect was scrutinized intently to determine if it was in fact the problem. Only if it proved not to be did the examiner back up and start along another path. In this manner, the experts appeared to use a depth-first search method, and, to emulate them, so does the RTFM system.

The code which implements this critical aspect of the system can be found in frfun.1 when we define the daemon that watches the state slot of a node. We see in this procedure that when the daemon is activated, if the node’s query slot is not nil then the daemon adds the query and links slots of the node to the queries and links slots of memory:

```
... (if (not (equal nil (query $self)))
  (progn
    (assert-val {mem} 'queries (cons (query $self) (queries {mem})))
    (assert-val {mem} 'links (cons (links $self) (links {mem}))))
...)
```

Here we clearly see the stack-like behavior as the new question and links are pushed onto the top of the memory’s question and links lists.

As a performance example, refer back to the kit sub-knowledge base in chapter 3.3. Were a student to reach this node and answer “No” to the question, we would see the three nodes nodekittest, nodekitbranch, and nodekitinfo be activated. Any information from any of these would be immediately displayed; however, after that RTFM would follow the branch in reverse order (because of the stack) beginning with nodekitinfo until it was completed. Afterwards, RTFM would return to follow the
nodekitbranch branch, and then, once that was completed, the nodekittest branch would be explored.

4.2.2 Multiple Answers

An additional function of the RTFM system is that the supervisory engine is capable of accepting more than one answer to a question. This allows users of the RTFM system to allow themselves the opportunity to explore more paths through the knowledge network of the system. This option was implemented because it was recognized that a problem is not always obvious and thus it might be desirable for users to indicate which problem they think is most affecting them (by listing it first), but also include a number of other answers to a question in their response. Thus, a user can use RTFM to provide knowledge on the most likely problem, but, should that fall through, the user is not required to reset the system and start over; RTFM can easily keep track of all the potential sidebranches and then explore secondary lines of questioning if necessary.

This option is enacted by answering any given question with a series of answers, separating multiple answers with an “&” symbol. Thus, given the following question:

The RTFM system has information on a variety of chips, particularly on those given out with the 6.111 lab kit. Please enter the chip you wish to get information for (ex: ‘LS04’, or ‘20v8’) and any available information will be displayed.

Which chip would you like information for:

==> 

One could answer with a simple request for a single chip:

==> LS163

Or, one could answer with a request for several chips:

==> LS163&LS151&29C10
The first answer listed is the path that you will follow first\(^3\) and the paths proceed from left to right. Some care should be taken when using this option as it can affect the search path one takes through the RTFM knowledge base — remember that RTFM uses a depth-first search algorithm to provide information and so the first answer in a series should reflect the initial path you wish to investigate.

To follow up on the pattern matching behavior as described earlier in section 3.2, this function allows users to try different pieces, even "overlapping" pieces, to match their digital problem (see Figure 4-2).

\(^3\)However, immediate information is presented for all answers in reverse order. So for the example, the initial information about each chip would be presented immediately beginning with the 29C10, followed by the LS151, and then the LS163. However, the questions and continued path of the LS163 would be followed first.
Chapter 5

RTFM User’s Guide

This chapter is intended to also be used as a separate document describing the use of the RTFM system to potential users.
5.1 Overview

The RTFM digital expert system is designed to help students gain a better understanding of the labs and projects they are asked to build by helping them to debug and design those projects. In many cases this is accomplished not by giving students solutions to their problems, but rather by trying to give them some information and then pointing them in the right direction to find the answer. 6.111 is a lab class, and as such it is hoped and expected that you will explore your own projects and designs to gain a greater understanding of digital design. However, it is also recognized that you will occasionally get stuck; RTFM is intended to help you at these times.

To use the RTFM system, you need merely to login to either Athena or the Sunpal (6.111 lab) system. If on Athena, you will need to setup the 6.111 locker¹ (on Sunpal, the 6.111 locker is added during login):

% setup 6.111

This being done, you may begin an RTFM session by typing 'RTFM':

% RTFM

This should drop you into a Common LISP shell and load all the files needed for RTFM. Be patient, some of them take a little while. After loading, the RTFM system will automatically be activated and you will begin your session. When you are

¹You may use the add command, but you will also need to source the .attachrc to setup the environment appropriately. The setup command automatically sources this file.
done and quit the RTFM program, you will be returned to your normal environment automatically.

If, for some reason, RTFM should halt in its operation and you find yourself in the Common LISP shell, you may try to restart RTFM by entering the command:

> (expert)

Alternately, you may quit from the Common LISP shell with the command:

> (bye)

5.2 Inside RTFM

Inside the RTFM system, you ought to be guided by a fairly intuitive process. RTFM will ask you a number of questions to try and figure out how your system is misbehaving and what information might prove helpful to you. As you progress, RTFM will tell you more things and ask you more questions.

The questions should be answered using one of the prompted answers. These answers are usually contained in brackets (][')s in the question, though a few (particularly with chip information) are indicated in other ways. All are clearly indicated by RTFM.

However, in addition to the simple answers, two other options exist. First, you may answer with more than one answer. Instead of answering with a single answer, if you wish to pursue multiple knowledge lines you may answer with several of the prompted answers, separating each with an '§'. Do not include spaces in these multiple answers, only the '§' should separate answers. For example, if you reached the node which lets you get information on TTL chips and wanted information on several chips, you could enter:

===> 1s04&1s151&1s163

to get information on the LS04, LS151, and the LS163 chips. Be sure to give your most desired answer first, as this is the first branch which RTFM will follow.
As well, there are a few system commands to which RTFM will respond. These commands are:

- !help or ?: These commands will give you a simple help screen inside RTFM to help you get around better.

- !backup: This command will drop the current line of investigation and backtrack through the knowledge network to find the last sidebranch and start investigating it. Careful! This does not take you back to the previous question.

- !activate_: This command lets you activate parts of the knowledge base if you know the name of the node you wish to activate. For the most part this is not geared for user use, but it can allow quick access to some nodes, particularly chip nodes which are named “node<chipname>”. Thus, if you wanted to immediately get information about the LS393 chip, you could tell RTFM: !activate_node<ls393>.

- !quit: This command immediately ends the current session of RTFM. You will be asked if you wish to exit RTFM or start a new session.

With these options available, you ought to be able to use RTFM to aid your project design and debugging. Good Luck.

5.3 Example

All right, now that you have a sense for the RTFM system and some of its features, we are now going to look at an example.

Imagine that we have a problem debugging an FSM. In particular, we have written an FSMC file describing the FSM’s behavior; however, palasgn gives us the following messages when we run it:

```% palasgn test
Trying heuristic 0...```
>>>AssignCombOut: cannot allocate pin for output p18, abort

Trying heuristic 1...

>>>AssignCombOut: cannot allocate pin for output p18, abort

Trying heuristic 2...

>>>AssignCombOut: cannot allocate pin for output p18, abort

Trying heuristic 3...
Using heuristic 3 is at your own risk !!!
The output file(s) may contain errors.

File test.pal
%

Not knowing what is going on, we decide to ask RTFM for help; beginning at the athena prompt, we first invoke RTFM:

% RTFM
<CLISP is executed, then loads a bunch of files...>

Refreshing knowledge base and memory......Done
Welcome to the RTFM v1.0 system. This program is intended to help you to debug and design your digital lab projects, particularly those for 6.111.

Any questions, comments, bug reports, or suggestions should be sent to owenwess@mit.edu

Thank you for using the RTFM program, and I hope you find it helpful.

copyright Owen B Wessling, 1995. All rights reserved.

=================================================================

The RTFM system works by asking you to identify what sorts of problems you are experiencing and then directs its knowledge activation according to your answers. As such, answer any questions to the best of your ability at the time. Feel free to send any comments about confusing sections to the above individual.

Also, keep in mind that this is a tool for educational purposes. It is not designed to give you answers to your problems, but rather to give you sufficient information that you can either deduce the solution or find the problem and correct it.

=================================================================

Now, which of the following areas do you need help in:
- You need information about a specific [c]hip.
- You are experiencing problems with the Nubus lab[k]it.
- You need [d]esign help.
- You have a problem using a particular (software) [t]ool.
- You need general help debugging your [s]ystem.

=>

We believe we want to explore the tools option, but figure it might also be a
problem with the 20v8 we are using, so we decide to add the *chips* option as a secondary path:

```
==>
```

We have information on most of the software tools used for 6.111, with which of them are you having a problem:

[R]educe
[P]al
Palas[g]ln
[F]smc
Fsm[d]
[A]sem
Pro[m]dio

```
==>
```

While we are using FSMC, our problem has occurred with the actual use of Palasgn, so we choose that as our primary answer, but again, FSMC might be the originator of the problem, so we add that as a second option:

```
==>
```

Which of the following error messages most closely matches your message or problem:

```
Trying [H]ueristic 3...
Warning: foo cannot be [a]ssigned as specified
Cannot allocate [p]in for <bartype> output foo, abort
```

```
==>
```

All right, Palasgn gave us two errors, it clearly couldn’t allocate a pin for p18, and that is what seemed to be failing in each heuristic, so we decide our first answer is 'p'; however, it did run out of heuristics, so we decide to include 'h' as well:

```
==>
```

Palasgn has several methods for allocating pins to PAL output pins, each of which places a different degree of importance upon the user’s specifications in the .eqn file. Successive heuristics pay less attention to the user’s request for a particular pin having a particular signal, and pay more attention to just getting all the information in the PAL.

Heuristic 3 is Palasgn’s last ditch effort to get it all in there. In general, any output from Heuristic 3 will have some errors. Sometimes, there will be no real output with the .pal files being blank; sometimes a few signals have been arbitrarily left out; _occasionally_ everything will be fine. Generally, if you hit Heuristic 3, check over the PAL files _very_ carefully before using them.
Now, why might it have to resort to H3? Usually, H3 is encountered because you are trying to jam too much information on too few PALs. Count your number of outputs you have, and make sure your PALs have at least that many output pins (always necessary). Also, check your pin equations and your device type. Most PALs have a limited number of summation terms that they can handle. The 20v8, for example can't handle anything over 8 terms on any of its pins. The 22v10 fares better (take a look at its data spec) and can handle more terms on some of its pins. Lack of output space, whether for raw output numbers, or for pins that can handle the equations, is the primary cause for reaching, and failing, Heuristic 3.

Palasgn cannot allocate a signal that you need to _any_ pin in the PAL. This is, of course, a problem, and one that often enacts successive heuristics, and often results in failing to deliver a usable .pal file.

Basically, you are trying to jam too much information in too small a place. Count the number of outputs you have, and make sure your PAL(s) have at least that many output pins (always necessary). Also, check your pin equations and your device type. Most PALs have a limited number of summation terms that they can handle. The 20v8, for example can't handle anything over 8 terms on any of its pins. The 22v10 fares better (take a look at its data spec) and can handle more terms on some of its pins. Lack of output space, whether for raw output numbers, or for pins that can handle the equations, will often result in a signal not being allocatable, so check these areas for problems in your design.

FSMC is a complicated program and it is HIGHLY recommended that you read the section of the FSM Design Tools Handout that deals with FSMC. Doing so will greatly decrease the number of errors you have in your .fsm files.

Also, note which line FSMC indicates has a problem. Most problems in FSMC are attributed to the right line, so as you learn more about the problem, look for the error there first.

Select the error message that FSMC is giving you: 
[M]issing operator 
[L]abel: ''foo'' referenced but never defined 
[I]mproper statement termination 
[O]ther

This is when we begin to understand the problem. RTFM says that a 20v8 (the kind of PAL we are using) can only handle 8 terms; we check our test.eqn file and realize that the p18 output has 10 terms. That would clearly be the problem. Not knowing what the other differences between a 20v8 and a 22v10 are, we decide to backup the RTFM system, to the chips section (the first sidebranch we indicated), and get some information about those chips before ending this session:

```bash
!backup
```

The RTFM system has information on a variety of chips, particularly on those given out with the 6.111 lab kit. Please enter the chip you wish to get information for (ex: 'LS04', or '20v8') and any available information will be displayed.

Which chip would you like information for:

```bash
22v10\&20v8
```
The 20v8 is a standard PAL chip, used to implement control signals and FSMs. Note that pins 1 and 13 are used for the CLK and output enable of any registered outputs. Also realize that combinational outputs are _not_ tristated, so the /OE should have no effect on them. If no registered outputs are used, pin 13 may be used as another input. Also, keep in mind that the 20v8 can only handle a maximum of _8_ terms in the sum of its output pins. If no registered outputs are used, pin 13 may be used as another input. Further information may be retrieved if you are having specific problems with your 20v8.

The 22v10 is probably the most common PAL used in 6.111 as it has the largest number of possible outputs as well as the capacity to handle large sums. If you have any registered outputs, you must provide your system CLK to pin 1, otherwise, you may use pin 1 as an ordinary input. The 22v10, unlike the 20v8, can handle more than 8 terms in most of its output sums. For each of the output pins, the number of sum terms it can handle is indicated in parentheses. Tristate outputs are controlled by internal programming, and must therefore be controlled in your .PAL file. No external pin is used to control tristatability with either registered or non-registered outputs.

Further information can be retrieved if you have specific problems with your 22v10.


With that information, we feel confident that we can fix our problem and also handle any other modifications necessary to convert our test.eqn file from using the 20v8 chip to using the 22v10, thus we end our RTFM session:

```plaintext
!quit
```

RTFM has depleted what it considers to be relevant questions. Please take the time to think about your problem and try to figure out why your problem exists. If you are still stuck ask a TA for help.

Do you wish to quit the RTFM system [y/n]: y

Bye.
5.4 Bugs and Exhaustion

If you encounter any bugs in the system, or you find RTFM exhausts itself before giving you enough information, send email to whomever is listed at the beginning of the program. RTFM is an evolving program, so your comments and discoveries are valuable to the system.
Chapter 6

RTFM Programmer’s Guide

This chapter is intended to also be used as a separate document describing the development of the RTFM system to potential RTFM programmers.
Welcome!

Welcome and thank you, for you have taken the first step in immortalizing at least some part of yourself so that other 6.111 students may benefit from your experience. That is, you have either already considered, or are now about to consider, the possibility of updating and adding to the knowledge base in the RTFM Digital Design expert system. If you are not already familiar with the RTFM system, I suggest you play with it a little first. Get to know it, learn something about its strengths and weaknesses, what it knows and doesn't know. After that, come back here and we'll tell you a little about how to go about augmenting the knowledge base.

Ideology

The thing I ask that anyone playing with the RTFM system keep in mind is that this system is designed to help students learn. It is not intended to solve all their problems and to give them every answer whenever they need it. That doesn't inspire learning or thinking; that inspires memorization and regurgitation. We want students to learn digital design and debugging, so if they need help, give them enough information to figure out the answer and point them in the right direction. Metaphorically, if they're thirsty, don't show them the oasis, just draw them a map.
6.3 What You Will Need.

If you plan to play with the RTFM system and update its knowledge base\(^1\) you will need to get a number of files from the 6.111 locker. These files are in the /mit/6.111/RTFM/ directory. Namely, you will need all the files in the frobs\(^2\) as well as the files in the sources directory. You may either transfer these directly, or you may acquire them via a tared file in the /mit/6.111/distribution directory\(^3\) called RTFM.1.tar. And you'll need a CLISP editor. Almost any flavor of Common LISP may be retrieved from the Common Lisp Repository, which is accessible via anonymous ftp at ftp.cs.cmu.edu in the /user/ai/lang/lisp/ directory. Odds are likely that you may also find a copy suitable to your needs in the .../RTFM/clisp directory. You will only really need copies of the sources files (they're the only ones you will change), though getting the others into a convenient place might make things a little easier to work with.

6.4 What Should You Change?

Acquire what files you need. To modify the knowledge base, you are going to change or augment the files kit.1, system.1, chips.1, design.1, tools.1. Each of these files contains a separate part, or sub–base, of knowledge for the RTFM system:

- **Kit.1** contains the portion of the network which relates to the Nubus lab kits which the students use. It is here that information regarding how those kits work, and how to fix those kits when they don’t work, is kept.

- **System.1** holds some loose information about how to debug certain paradigms of digital design. This is probably the most general and loosely defined area of the knowledge base.

\(^1\)If you're interested in playing with the supervisory engine, take a look at the M.Eng Thesis document on RTFM by Owen Wessling ’96. There is room for more features and system commands, which you are welcome to make if you so desire.

\(^2\)You will not need to change these, but you will need them to load the frobs platform into Clisp

\(^3\)This distribution directory is also accessible via anonymous ftp at sunpal2.mit.edu

53
- **Design.1** is the part of the network which deals with the design of digital systems, making something of a pre-emptive strike on the debugging process. ("What’s the best way to fix a bug?" "Don’t have a bug.") This part of the knowledge base is probably the most critical in terms of making sure you’re not giving away an answer. Remember, the idea of RTFM is to help students think and learn, not do the class for them.

- **Chips.1** is a large sub-base which contains information about the chips (mostly TTL) which a student is likely to encounter in 6.111. The knowledge contained here is of two main types: 1) a chip pinout is offered because many bugs are because of miswiring or inversion errors; 2) some of the common misuses of the chip may be described. This is not supposed to be a copy of the specification sheet — otherwise there’d be no reason to give them such sheets...

- **Tools.1** attempts to provide information about the various software programs available to 6.111 students.

You should feel free to change and/or augment these files as you see fit, though I do ask that you try to maintain the sub-base abstraction areas. Also, if you think an entirely new sub-base needs to be defined, go for it, just write up a brand new file⁴.

When you make a new node you will need to integrate it into the appropriate sub-base. There are a number of dependency-like diagrams which show something of the interconnectivity of the knowledge network shown in Figures 6-1, 6-2, 6-3, 6-4, and 6-5. Find the place you wish to insert it and change part of the preceding node’s query slot to ask a question which would divert the network to your node; then add your answer and the name of your node as a response-link sublist in the *links* slot (more on what these slots are later). You may also need to add questions to your node to redirect the flow back to the original network; again, you can do this with the *query* and *links* slots of your new node.

---

⁴If this is the case, you will probably need to make a small change to RTFMinit.1 in the node *nodeRTFMstartup* to allow for the branch into the new sub-base.
Figure 6-1: Chip Sub-Knowledge Base Network

Figure 6-2: Kit Sub-Knowledge Base Network
Figure 6-3: Tools Sub-Knowledge Base Network

Figure 6-4: System Sub-Knowledge Base Network
6.5 What is a Node

Each node has four slots of information to which you should be concerned: the \textit{id}, the \textit{info}, the \textit{query}, and the \textit{links} slots. The \textit{id} slot is both the name of the node and the identifier of the node; it is normally just a symbol. The \textit{info} slot contains information (surprise); it should be a list of strings which contain the information to be passed on to the user, or \texttt{nil}. Similarly, the \textit{query} slot contains the question of the node and should also be a list of strings or \texttt{nil}. The \textit{links} slot should contain a list of lists. Each sublist should be a response–nodes set in which the first element of the list is a response to the question in the \textit{query} slot. The remaining elements of the sublist are the nodes to be activated if the user makes that response. A recap of this information is in Table 6.1. Take a look at any of the subbase files for examples.

As well, each node has a slot which you shouldn’t be concerned about, but should know how it works: the \textit{state} slot. This is essentially a binary bit which indicates whether the node is active. Normally, a node is \texttt{inactive}; when a user responds in
a described manner (referring to the answer to a previous node’s query), a node may be activated by the RTFM system. When this happens the information in the info slot is presented and the question and links in the query and links slots are pushed onto a memory stack. The RTFM system works by popping this memory stack and asking the directed question and then activating the prescribed nodes.

6.5.1 A Few Tricks

Now, there are a few fine lines to understand about the system, as not all is as clear as it would seem. There are two main things to realize about the system:

1. There is a fine line between the information slot and the query slot. RTFM is set up so that, in fact, a query slot can contain straight information and not ask a question. That is, if the links slot is nil, then RTFM will print the strings from the query slot, but not pause for a response from the user. Thus, the info slot and query slot become remarkably similar. However, a fine difference does exist! The strings in the info slot are printed to the screen as soon as the node is activated; however, the strings in the query slot are printed at the delayed time when the node is popped from the stack. This is a subtle difference, but it allows for more freedom in creating the knowledge base.

2. The other detail is to recognize that more than one node may be activated from a particular response. That is, the list (d {nodeA} {nodeB} {nodeC}) is a perfectly good sublist for a links slot. What you need to be aware of is how those nodes will be activated. Namely, that the nodes are activated in order, and that means the info slots will be printed immediately for all three and in the order of appearance (here: A, B, C), but that their query and links slots will be pushed onto the stack in order, so that they will be processed by RTFM in reverse order. Thus, nodeC is the first branch to be explored.
6.5.2 How to Create a Node

Now that you know what a node is, all that is left is to tell you how to make one. Making a node is rather easy — all you need to do is tell the CLISP buffer (after you’ve loaded the frfun.1 file) to make a node using the following command:

> (make-node _id _info _query _links)

Now, this command is fairly intuitive and matches closely with what is stored in the node. Each argument is assigned to the appropriate slot, and the state slot of the node is set to be inactive. For example:

```
(make-node 'foo
    ("This is the information slot")
    ("Is this the Query slot \[y/n]\? ")
    ((y {bar}) (n {baz}))
)
```

Execution of this line would create a node named foo, overwriting any existing node named foo. Upon foo’s activation, RTFM would tell the user: This is the information slot. When RTFM popped foo’s query and links slots from memory, it would ask the user: Is this the Query slot [y/n]?, to which RTFM would expect an answer of either y or n. When the user responded with one of those, RTFM would activate the node bar or the node baz, respectively. RTFM will compare responses to its questions to the links list to try and recognize answers; if it doesn’t recognize an answer it will try to process the response as a system command; if this fails, RTFM will tell the user that the response is not recognized. In either of the last two situations (a system command or an invalid response) RTFM will also repeat the question.

6.6 Parting Words

Good Luck. You now have all the information you really need to augment the RTFM knowledge base. I have tried to keep this programmer’s guide short and to the point
in the hopes that more people will read it and thereby play with the RTFM system. If you find you want more information, or are particularly interested in the supervisory program of the RTFM system, you should feel free to look at the source code (I tried to keep it pretty well documented) as well as the M.Eng thesis for which this program was written. Again, good luck, and may RTFM flourish.
Chapter 7

Conclusions

The RTFM system is decidedly capable of alleviating all of the problems which it was designed to help; however, the system will require some time for it to fully mature. Recall that the RTFM system has been designed to help those who are inexperienced with the techniques of digital design to debug and design their projects. The primary goal is to help students learn and think about the digital design process; RTFM does this by satisfying several subgoals:

- The RTFM system is capable of helping with many of the more common problems that students encounter. True, the problems it can help with are of a more trivial nature, but it is still these problems which make up the bulk of the problems students have; thus the RTFM system can help them.

- In a complementary fashion, because RTFM can handle the more simplistic problems, the teaching assistants are allowed to spend more time helping students with the harder and more complicated problems.

- By acting as a repository for knowledge, the RTFM system reduces the ramp-up time for new assistants. This helps the assistant by boosting his level of ability which is a benefit to both the assistant and the students he helps.

- RTFM is designed to present the process of digital design as well as raw information. This insight helps students both directly and indirectly in that it can
help a student track a simple bug; and later, the student will have a better grip on the process for dealing with a more complicated problem.

- And, of course, the RTFM system is a program and, therefore, it is always available to students.

The RTFM system succeeds in all of these things, and because of that, it meets its primary goal — to help students to *learn and think* about their digital design projects.

The RTFM program is just that — a program; it can be used anytime, anywhere as long as the student has access to a computer on which it can run. Furthermore, the program has been designed such that any Common LISP derived language should be able to read the files and act as a host for the RTFM system. In this manner, RTFM can provide its services whenever a user is in need.

The RTFM system has the knowledge base to handle many of the common problems encountered when one begins to study the process of digital debugging. While the system is currently designed to deal particularly with problems seen in the 6.111 *Introductory Digital Systems Laboratory* course at the Massachusetts Institute of Technology, many of those problems are of a general nature, and the program is designed to grow. As more people use the system and the users and maintainers see new problems and different aspects of the older problems, the knowledge base of the RTFM system will grow and the functionality and capability of the system will expand. As it stands now, RTFM can help users design and debug the simple-to-moderately complex projects one encounters when first learning digital design; as it is given more knowledge, that capacity will increase.

The RTFM system also acts as a knowledge repository. Indeed, a relationship which cycles from a new teaching assistant learning from the RTFM system to that same assistant adding new information to the RTFM system is likely to be synergistic. When that assistant examines the system and sees what it contains, he not only learns more about how to help students, but he also learns what the RTFM system lacks and can direct his energies to providing that knowledge. Thus, as the process continues, as more and more assistants move through the class, the RTFM system will provide
a valuable service to them, as well as receive valuable knowledge from them.

The knowledge that the RTFM system does contain allows it to help students with many of the more common problems. This helps in two ways: the students benefit by having a constantly available source of help for those simple problems. And, by freeing the TAs of those simpler problems, RTFM allows them to concentrate their help on the more complicated problems that arise.

How the RTFM system will fare in the years to come is still uncertain and unknown, though the future holds great promise. However, in its original endeavors, the RTFM system meets with complete success.
Appendix A

Knowledge Base Code Listings

The following source code forms the knowledge base of the RTFM system. This code may be retrieved via anonymous ftp from sunpal2.mit.edu:/pub/digital_tools/. These files, as well as the supervisory engine files, are contained in the RTFM.1.tar archive file.
A.1 Chips.1

; File: chips.l
; Description: Knowledge base for the RTFM 6.111 Lab expert
; Expansion: KCL version
; Author: Owen Wessling
; Last Mod: 9/17/95 v1.0

This file is a part of the RTFM 6.111 Lab Expert project.
This is a part of the M.Eng thesis by Owen Wessling, M.Eng EECS '96, MIT.
Copyright (c) 1995 Owen Wessling All Rights Reserved.

This software is supplied "as is," without any kind of warranty whatsoever, but not limited to the implied warranties of merchantability and fitness for a particular purpose. The author and MIT make no representations about the suitability of this software for any purposes and do not promise to support it in any way.

This software is not to be incorporated into a commercial product, or access to it sold, without written permission from MIT. Otherwise, copies which include this notice may be freely made and distributed.

This file contains the node descriptions of the chip section of the RTFM 6.111 Lab expert. It attempts to provide relevant information for all chips students are likely to encounter in 6.111.

At present (9/17) this file is intended to have information on the following chips:

LS: 00,08,04,06,10,14,20,28,42,74,85,86,115,123,138,150,151,161,163,169,194,244,245,257,259,283,374,377,393
PALs: 20V8, 20V10,
SRAM: 6116
A/D: AD558, AD670
Flash: 2BF256A
UART: AY3-10150
uCont: AM29C10A
XTAL: 4-pin

Just to clarify how to make a node in the knowledge base:

(make-node _id _info _query _links) is the command to create a node

_id is the name and identifier of the node
_info is a "list" of strings containing information to be printed when the node is activated
_query is a list of strings containing a question. if the node is a dead-end then_query should be nil
_links is a list of response—node lists. Each response—node list is a list of the form (response node1 node2 ...) where node1, node2, etc are the nodes to be activate if _query is answered with response.

Note: most chip information is a dead end —— once info on a chip is given there's little else to pursue from it. Of course, other topic lines could be activated from prior questions.
The RTFM system has information on a variety of chips, particularly on those given out with the 6.111 lab kit. Please enter the chip you wish to get information for (ex: 'LSO4', or '20v8') and any available information will be displayed.

Which chip would you like information for:

1. LS00
2. LS02
3. LS04
4. LS08
5. LS10
6. LS14
7. LS20
8. LS32
9. LS42
10. LS74
11. LS85
12. LS86
13. LS112
14. LS123
15. LS138
16. LS150
17. LS151
18. LS161
19. LS163
20. LS169
21. LS194
22. LS244
23. LS245
24. LS257
25. LS259
26. LS283
27. LS374
28. LS377
29. LS393
30. 20v8
31. 22v10
32. 6116
33. HM6116P
34. AD558
35. AD670
36. 28F256A
37. FLASH
38. AY3–10150
39. UART
40. AM29C10A
41. 29C10
42. XTAL

(make-node 'nodechipselect nil

"The LS00 is a quad 2-input NAND gate."

+-------+ For each group Y = NAND(A,B)."
| 1A|1 14Ivcc |
| 1B|2 L 1314B |
| 1Y|3 S 1214A |
| 2A|4 0 1114Y |
| 2B|5 0 1013B |
| 2Y|6 913A |
| GND|7 813Y |

+-------+ Otherwise, you may have a faulty chip."
nil
nil

(make-node 'nodeLS02
" The LS02 is a quad 2-input NOR gate.
" 1Y1 14Vcc "
" 1A1 1 131/4Y "
" 1B1 3 S 12I4B "
" 2Y1 4 0 1114A "
" 2A1 5 2 1013Y "
" 2B1 6 913B "
" GND1 7 813A "
" "
"
)
il
nil

(make-node 'nodeLS04
" The LS04 is a Hex inverter chip.
" 1A1 1 14Vcc "
" 1Y1 2 L 1316A "
" 2A1 3 S 1216Y "
" 2Y1 4 0 1115A "
" 3A1 5 4 1015Y "
" 3Y1 6 914A "
" GND1 7 8134 "
" "
"
)
il
nil

(make-node 'nodeLS08
" The LS08 is a quad 2-input AND gate.
" 1A1 1 14Vcc "
" 1B1 2 L 1314B "
" 1Y1 3 S 1214A "
" 2A1 4 0 1114Y "
" 2B1 5 8 1013B "
" 2Y1 6 913A "
" GND1 7 813Y "
" "
"
)
il
nil

(make-node 'nodeLS10
" The LS10 is a triple 3-input NAND gate chip.
" 1A1 1 14Vcc "
" 1B1 2 L 1311C "
" 2A1 3 S 1211Y "
" 2B1 4 1 1113C "
" 2C1 5 0 1013B "
" 2Y1 6 913A "
" GND1 7 813Y "
" "
"
)
il
nil

(make-node 'nodeLS14
" The LS14 is a Hex Schmitt-Trigger Inverter chip.
" 1A1 1 14Vcc "
" 1Y1 2 L 1316A "
" 11111 "
" "
"
)

The LS02 is a quad 2-input NOR gate. For each group Y = NOR(A,B).

If you are having a problem with this chip, check your power and ground pins and verify proper levels. Otherwise, you may have a faulty chip.

The LS04 is a Hex inverter chip. For each group Y = /A.

If you are having a problem with this chip, check your power and ground pins and verify proper levels. Otherwise, you may have a faulty chip.

The LS08 is a quad 2-input AND gate. For each group Y = AND(A,B).

If you are having a problem with this chip, check your power and ground pins and verify proper levels. Otherwise, you may have a faulty chip.

The LS10 is a triple 3-input NAND gate chip. For each group Y = NAND(A,B,C).

If you are having a problem with this chip, check your power and ground pins and verify proper levels. Otherwise, you may have a faulty chip.

The LS14 is a Hex Schmitt-Trigger Inverter chip. For each group Y = /Schmitt(A).
If you are having a problem with this chip, check your power and ground pins and verify proper levels. Otherwise, you may have a faulty chip.

For those not familiar with a Schmitt-Trigger, the purpose of the device is to sharpen edges on a 'noisy' line.

The LS20 is a dual 4-input NAND chip.

The LS32 is a quad 2-input OR chip.

The LS42 is a 4-line BCD to 10-line decimal decoder chip.

The LS74 is a dual DFF, positive-edge triggered.

Note that the /PRE, and /CLR pins are asynchronous.
The LS85 is a 4-bit magnitude comparator. This chip is intuitively obvious given that $A \# B$.

If you are using it to identify cases when $A = B$, check that your input codes ($A > Bin$, etc) fit your design.

The LS86 is a quad 2-input XOR gate.

For each group $Y = XOR(A, B)$.

If you are having a problem with this chip, check your power and ground pins and verify proper levels. Otherwise, you may have a faulty chip.

The LS112 is a dual NEGATIVE-edge triggered JKFF. Note that /PRE and /CLR signals are asynchronous, and also that both be brought LOW together, both $Q$ and $\overline{Q}$ should go HIGH, however, this output is not guaranteed. Note also that PRE and CLR are ACTIVE LOW.

The LS123 is a dual retriggerable monostable multivibrator. Very useful for debouncing signals, problems with the LS123 are usually associated with the resistor and capacitor values used. Also note that the CLR signals are ACTIVE LOW.
The LS138 is a 3-line to 8-line decoder.

Note that the output lines are held HIGH by default and only the selected line is brought LOW. Also note that the LS138 has 2 enable pins, G1 and /G2 (/G2 is enabled by either of its pins), and that G1 is ACTIVE HIGH and that G2 is ACTIVE LOW.

The LS150 is a 4-bit/16-line multiplexor.

Note that the output W is the complement of the data line selected. That is:

W = /Exx

Also note that the enabling pin G is ACTIVE LOW.

The LS151 is a 3-bit/8-line multiplexor.

Note that the output of the LS151 can be either the selected signal or its complement:

Y = Dx  W = /Dx

Also note that the enabling pin G is ACTIVE LOW.

The LS161 is a synchronous 4-bit counter with ASYNCHRONOUS Clear and Load.

If your having problems counting at all, check that your /CLR and /LOAD signals are correct. Also, both ENT and ENP must be high to enable counting.

If counting is occurring, but not at a proper pace, you may have a problem with your reset rates. Also, if you are cascading multiple counters, you should be careful how you enable your '161s -- in that RCO is only valid while you will have problems coating with multiple chips if you hold ENT high and pulse ENP. Also realize that the asynchronous clear or load will effectively subtract one from your counting pattern, so check your reset number if your count is off by exactly one.

If your counting is occurring, but not at a proper pace, you may have a problem with your reset rates. Also, if you are cascading multiple counters, you should be careful how you enable your '161s -- in that RCO is only valid while you will have problems coating with multiple chips if you hold ENT high and pulse ENP. Also realize that the asynchronous clear or load will effectively subtract one from your counting pattern, so check your reset number if your count is off by exactly one.
The LS163 is a synchronous 4-bit counter with SYNCHRONOUS CLEAR and Load.

If your having problems counting at all, check that your CLR and LOAD signals are correct. Also, both ENT and ENP must be high to enable counting.

If counting is occurring, but not at a proper pace, you may have a problem with your reset rates. Also, if you are triggering, check that your ENT control allows for proper triggering.

The LS169 is a synchronous 4-bit up/down counter.

If counting is occurring, but not correctly, make sure your U bit is correct. If the count is off by exactly one, you may want to check your reset circuit, or, if the reset is RCO triggered, check that your ENT control allows for proper triggering.

The LS194 is a 4-bit bidirectional shift register.

Leaving them unconnected is a common problem which results in unpredictable, but usually erroneous output.

The LS244 is a tristate octal buffer and line driver.

Extremely useful for controlling bus access (i.e. bus contention is _bad_).
If your '244 is not driving your line correctly, make sure that your control signal is ACTIVE LOW, like the enabling pins. Otherwise, any problem with the '244 is likely caused by inappropriate chip power.

The LS245 is a tristate octal bus transceiver. That means you can send information in either direction, unlike with the 244.

The LS257 is a quad 2-1 multiplexor. Note that your enabling pin, G, is ACTIVE LOW.

The LS259 is an octal (addressable) latches chip. This chip combines several functions and care should be taken to verify your use of the control pins, /CLR and /G. Used correctly, the '259 can act as an 8-line demultiplexor and as a settable, though simple, memory device.

The LS283 is a 4-bit full-carry adder.
Make sure that your chip is wired in the correct direction," that is, that your 4-bit numbers are correctly aligned from LSB to MSB. Also make sure that your carry input is connected (either to the right chip or to 0/1)."

If the output doesn't seem like it is the correct addition check your data bits a second time, and if not that, check to make sure the chip is getting appropriate power levels."

The LS374 is an octal D-type Flip-Flop."

Be aware of the unusual input/output mapping of this chip.

Also note that the D Flip-Flops of the 374 are positively edge triggered (as per normal D-type FF's)."

Note that the output enable is ACTIVE LOW. Otherwise," problems are likely to be caused by an inactive, or" glitchy clock signal." Note that the control pin /OE controls the output of" the FF's, not whether they are enabled. If you want to control the enabling of the DFF's reading of data, you want to use the LS377."

The LS377 is an octal D-type Flip-Flop with enable."

Be aware of the unusual input/output mapping of this chip.

Also note that the D Flip-Flops of the 377 are positively edge triggered (as per normal D-type FF's)."

Note that the enable is ACTIVE LOW. Otherwise," problems are likely to be caused by an inactive, or" glitchy clock signal." Note that the control pin /G controls the ability for the" DFF's to _read_ data. If you want to control the output" of the Flip-Flops, you want to be using an LS374."

The LS393 is a dual 4-bit binary counter."

Note that in this counter, your CLR control signal is ACTIVE HIGH, unlike just about every other counter."

Problems with the counter being paralyzed at 0 are likely" to be because the CLR is thought to be active low when it's not. Otherwise, problems counting are probably due to" a glitchy clock, though insufficient power may also cause sporadic errors."
The 20v8 is a standard PAL chip, used to implement control signals and FSM's.

Note that pins 1 and 13 are used for the CLK and output enable of any registered outputs. Also realize that combinational outputs are _not_ tristated, so the /OE should have no effect on them. If no registered outputs are used, pin 13 may be used as another input.

Further information may be retrieved if you are having specific problems with your 20v8.

The 22v10 is probably the most common PAL used in 6.111 as it has the largest number of possible outputs as well as the capacity to handle large sums.

If you have any registered outputs, you must provide your system CLK to pin 1, otherwise, you may use pin 1 as an ordinary input.

The 22v10, unlike the 20v8, can handle more than 8 terms in most of its output sums. For each of the output pins, the number of sum terms it can handle is indicated in parentheses.

Tristate outputs are controlled by internal programming, and must therefore be controlled in your .PAL file. No external pin is used to control tristatability with either registered or non-registered outputs.

Further information can be retrieved if you have specific problems with your 22v10.

The HM6116 SRAM is 2048x8-bit static RAM chip.

Note that all control signals are ACTIVE LOW.

Also, while it is not necessary, and your design may differ, we suggest connecting /CS to your system CLK.

Thus making sure the address bits are stable before letting the SRAM start its access.

Be sure to ground any unused address pins, otherwise you may not access the same place twice...
Further information can be found by specifying your problem.

If your problem with the 6116 SRAM: [R]ead from the SRAM
[W]riting to the SRAM

((r {nodeSRAMread})
(w {nodeSRAMwrite})

(make-node 'nodeSRAMread

"Bus contention, chip activation, correct signalling, and correct addressing"

"are all common problems encountered."

"Bus contention occurs where you have more than one device driving your bus. If"
"you have the input switches driving the same bus that the SRAM's 10 pins are"
"on, at the same time, that you want the SRAM to tell you what is in address"
"foo, the switches will mess with the signal and you won't get a valid number."
"This is what the LS244 and LS245 line drivers are for; you need to make sure"
"that only one thing is driving your bus at any given time."

"Another problem is chip activation. Remember that we suggest to connect /CS"
"to your system clock. This really shouldn't be a problem since you should"
"be reading data from the SRAM at the rising edge --- just after /CS was low,"n
"so the SRAM was active. However, if you're doing something with the data from"
"the SRAM in mid-cycle, you need to be careful of the /CS signal."

"Correct signalling is important --- remember that /WE and /OE are ACTIVE LOW."

"Finally, make sure your addressing is correct. Make sure any unused address"
"pins are grounded, otherwise you may float from one address to another when"
"you think you're addressing the same place."

"Another possibility is that if you are writing between reads, you may"
"accidentally overwrite some addresses if your address is not stable while"
"/WE and /CS is low. This is one reason that we suggest wiring /CS to CLK."

(make-node 'nodeSRAMwrite

"Problems writing to the SRAM usually occur not because the SRAM doesn't,"
"but because it doesn't write what you expect where you expect."

"Get a good analysis of what's going on --- wire your address to the Logic"n
"Analyzer as well as your control signals and your data bus. It's a lot"
"of wiring, but you usually need it to debug the write cycles."

"Now take a look at what is happening. Make sure that your address lines"
"are stable and the correct value (also be sure they're that way _before_."
"both /WE and /CS are low --- why?). Then, make sure the data is valid ---"
"the SRAM is transparent while both /WE and /CS are low, so be sure that data"
"is good until the end of the clock cycle."

"Of course, this doesn't cover the problem regarding invalid data on the data"
"bus; i.e. when your write looks bad, but in fact the SRAM write is fine, but"
"the data it's writing is bad."
"Is this what is happening to you? [y/n])

((y {nodebad databus})
(n {node designheuristic})

)
The AD558 is an 8-bit D/A converter. Note that this chip has unusual power and ground connections, as well as an analog ground.

This chip supports both a 0-2.56V range, and a 0-10V range. Please check that your connections between the 3 Vo's and the AGND match the desired mode, as shown in the data specifications.

Aso note that CS and CE are ACTIVE LOW.

The AD670 is an 8-bit A/D converter.

If you are having problems, verify that your input is as you expect and that you have wired the BPO and FORMAT signals appropriately.

Also check that your analog reference voltages are wired in accordance with the desired setup, as shown in the data specification sheets.

If you are having a problem during your read, especially with the particular symptom that you AD670 seems to be starting a new conversion, check that you drop /CE or /CS decidedly _after_ R-/W has been asserted. Otherwise there is a chance that the AD670 is seeing low /CE, /CS, R signals and starting a conversion.

Even better is to keep R-/W high normally and only bring it low when you wish to initiate a conversion.

The AD670 has two principle modes of operations:

- A [S]ingle Conversion, Single Read mode, and
- a [C]ontinuous Conversion, Single Read mode.

Some problems are particular to each of these modes, so to further examine any problems you may be having we must ask which of the above modes you are using:

1. Problems with the AD670 when used in Single Conversion mode may be caused by improper use of the /CE and/or /CS pins. Keep in mind that the AD670's output is only valid while _both_ of these pins are low. Indeed, both of these pins are normally tied together (and often tied LOW) as they almost always must be asserted in the same fashion for the chip to do anything.

As well, check you R pin. Single conversion is especially susceptible to the aforementioned problem of starting a conversion when trying to read if the R pin is not normally held high (indeed, if it is not, you are probably making the chip act in continuous conversion mode).

"Also, be sure that you wait for STATUS to drop back low (or wait the requisite 10 us conversion time) before looking at the data pins."
The most common error while using this mode is to not be sure that R is brought back high (in preparation for the read) while STATUS is high. This is imperative, as the AD670 stops with the next conversion after R has been brought high; however, if R is brought high when STATUS is low, another conversion will not be started, but the reset for the conversion for a new conversion will be in process --- thus any data generated will not be valid.

The 28F256A, or Flash, PROM is the standard PROM for 6.111. Note, like most chips, the control signals are ACTIVE LOW.

Vpp is used when writing to the PROM, as such, when using the '256 as a ROM, Vpp should be held LOW. Similarly, /WE should be held HIGH.

Most problems with the '256 occur for one of the following two reasons:
1) either not all address pins are driven; particularly those not used are left floating and this leads to inconsistent reads. Or, 2) either /CE or /OE is improperly driven. Both should be low for a data read.

Bringing either HIGH will place the output pins of the '256 into a tristate operation.

Another common problem is for a pin to bend up under the chip, so be careful when inserting it into your kit.

The AY3-10150 UART is a Universal Asynchronous Receiver and Transmitter. In other words you can use it to get or send serial data. Please read the data spec sheet if you have not, the complete operation of this highly complex device is beyond the scope of this program.

The UART is somewhat sensitive about the transmission/receiver clocks (TCP and RCP). These should be clocked at 16x the desired baud rate. Keep in mind that a little error can be OK, but much more than +/-5%, will probably cause a problem. Also keep in mind that +/-5% is often exceeded by a single count in 6.111.

DO NOT HOLD /RDAV LOW. Think about it. This is not going to cause correct operation....

Also, be sure your word bits are what you expect. The will be confused if you tell it there is no parity when there is. Check NP, TSB, NB1, NB2, and EPS. Also check CS.
The Am29C10A is a microprogrammer controller, which allows students to more easily build and use circuits to control projects using microprogramming.

Please review the data specification sheet for this part before building anything with it; the part is quite complex and trying to explain how to use it is beyond the scope of this program.

Problems common to the Am29C10A:

- Control signals are imperative with this device. Make sure you know what the signals do and how you intend to use them. Especially look at those signals you want to tie low (or high). Note that the CC iinput is ACTIVE LOW, make sure your input multiplexor agrees with that, or that you've inverted your condition code signals.
- Make sure your instruction word is correct. A lot of problems occur if your instruction word is backwards (particularly if you correct it on the analyzer pins).
- Also be aware of the nonstandard power and ground pins.
- XTALs are the easiest way to generate a system clock. The crystal oscillator produces a very nice waveform; however a few precautions should be taken and a few details known. Take a look at the output on an oscilloscope, notice that the XTAL rings pretty badly at each transition. This ringing should not be a problem, and should not cause a glitch in your clock signal. It will however do just that (which we will characterize as a bad thing) if you try to drive too much with the XTAL. The XTAL is not rated as a normal TTL chip --- it should only drive _1._ TTL input; but you can probably get away with 2 or 3 without seeing the glitch. Check it on the logic analyzer to make sure! Use the XTAL to drive either an inverter or a Schmitt trigger. The trigger is nice as it will clean up your clock signal, and can also drive multiple TTL inputs. An inverter will often work, as well. A good idea would be to combine the two --- drive the trigger with the XTAL and then use the trigger's output to drive _several_ inverter inputs. The resulting inverters ought to be able to provide a solid system clock to all the other chips in your system. If you need to divide your clock, do that between the trigger and the inverters.

A.2 Kit.l
This file contains the node descriptions of the kit section of the RTFM 6.111 Lab expert. It attempts to provide relevant information for problems students are likely to encounter with the NuBus lab kits of 6.111.

Just to clarify how to make a node in the knowledge base:

(make-node id _info _query _links) is the command to create a node

_id is the name and identifier of the node
_info is a "list" of strings containing information to be printed when the node is activated
_query is a list of strings containing a question. If the node is a dead-end then _query should be "("")
_links is a list of response--node lists. Each response--node list is a list of the form (response node1 node2 ... ) where node1, node2, etc are the nodes to be activate if _query is answered with response.

Just a column reference:

(make-node 'nodekitproblems nil
'"Okay, so you think something is wrong with your kit. First off, have you checked your kit to verify that the major components are working? Namely when you first got your kit, did you check to make sure all LEDs and switches worked and that power and ground were steady levels at +5, +12, -12, and 0 Volts at the appropriate places? [y/n]"
)'
((y {nodekitbranch} {nodekitinfo})
(n {nodekittest} {nodekitbranch} {nodekitinfo})
)
)

(make-node 'nodekittest
'"Kit testing is _very_ helpful. The kits used in 6.111 suffer a fair amount of abuse. They are durable and the systems are fairly robust; however, they can only take so much. Consequently, it is a good idea to check your kit"

"when you first get it to make sure all the major parts work okay. It is"
"much easier to have them fixed at the beginning of the class when you don't"
"have any wiring done than in the middle of a lab should you find a broken"
"component. As such, it is recommended that you perform the following tests"
"on your kit when you first get it, of course if you didn't do them when you"
"first got your kit, do them as soon as you can --- hopefully nothing will be"
"wrong, but better to find out now than after 8 hours of trying to figure out"
"why an LED doesn't turn on, only to figure out it's broken."

"o Test all your power voltage levels. Check that your analog voltages are" 80
"at their correct levels: -12, +12, and +5 Volts. Check that your analog" 80
"ground is a good steady 0 Volts. Minor deviations are okay (like 11.92" 80
"Volts instead of 12 Volts), but anything seriously off base should be" 80
"called to the attention of a TA or Arlin at the desk."

"o Check your digital power supplies. Make sure your you +5 is close (a" 90
"more deviation is acceptable here --- we need a TTL High level; anything" 90
"over about 4.5 Volts is fine. Again, check your ground, too."

"o Connect a fairly long wire to your now-verified digital +5 supply. Plug" 100
"the connection into each of your LED inputs and verify that each of them" 100
"work. IF any of them don't make a note of them and have them fixed (just" 100
"take your kit to the desk and tell Arlin what the problem is and he'll" 100
"fix it)."

"o Finally, take the supply end of that wire and plug it into each of your" 100
"switch connections (leaving the other end in a connection to a working LED)" 100
"and check out each of your switches. Make sure they all work."

"Now, if you find anything that doesn't work as you'd expect it to, either take" 100
"it to Arlin at the desk to be fixed, or talk to a TA. Anything that flat" 100
"doesn't work probably needs fixing, though a few power problems can be fixed" 100
"without desk intervention."

(make-node 'nodekitbranch nil
  '("Which part of your kit seems to be acting up:")
  '([P]ower problems
    ([L]EDs
      [S]witches
      [H]ex LEDs
      Logic [A]nalyzer related things (bus, connections, etc.)")
  )
)

(make-node 'nodekitpowerproblem nil
  '("Is your problem a [l]ack of power, or [w]rong power levels? You can determine" 130
    "Which of these is the case by looking at the four power LEDs in the top left" 130
    "of your kit --- if any of these are off when both switches for your kit are" 130
    "on (up) then you have a problem with lack _of_ power. If they're all on," 130
    "but your (for example) your +12 Volt supply is registering +8 Volts, then" 130
    "you have the wrong voltage levels. :"
  )
  '(([l]ackpower
    ([w]rongpower)))
)

nil
nil
nil
nil

nil
nil
nil
nil

nil
nil
nil
nil

nil
nil
nil
nil

nil
nil
nil
nil

nil
nil
nil
nil

nil
nil
nil
nil

nil
nil
nil
nil
(make-node 'nodelackpower

  "Lack of power to the kit is probably caused by a break in the connection"
  "to the wall current."

  "First of all, check the power strip that you've plugged your kit into (you"
  "did plug it in, right?) and check that the power strip is on. There is a"
  "switchbox at one end of the power strip with a light that will be on when the"
  "power strip is on. If that light is off, try flipping the switch, if it fails"
  "to come on notify a TA that your circuit breaker has been blown and you need"
  "it to be reset."

  "If your power strip is on, but some of your kit power LEDs do not come on,"
  "check first that _both_ of your power switches are on. Once you verify that"
  "both your switches are on, but one or more of your power LEDs is not, take your"
  "kit to the desk and have it fixed. If you feel like you can test the"
  "associated power strip to see if power is not getting through or if the LED"
  "is dead, but in any case something needs fixing so you'll need to take your"
  "kit to see Arlin."

  )
 nil
 nil

(make-node 'nodewrongpower

  "If the power strips of your kit do not have the appropriate values, check first"
  "that you are measuring them correctly. Namely, you are probably using the"
  "oscilloscope to check the power levels, and so you need to check that your"
  "scope knows the correct kind of probe you are using. Go to the CONFIG/CHAN"
  "menu and verify that your PROBE setting matches your probe (probably 10:1)."

  "If this doesn't 'fix' your problem, take your kit to Arlin (the desk) to have"
  "it fixed, something is wrong internally and probably needs replacing."

  )
 nil
 nil

(make-node 'nodekitLEDproblem

  "If you've checked the operation of your LEDs and verified that each comes on"
  "appropriately with a direct connection to +5 Volts, then your current problem"
  "is probably not with the LED but with the control of the LED."

  "Check the connection you've made to the LED and make sure it goes where you"
  "think it goes and make sure that the connection is good. After that, make"
  "sure that the pin you have it connected to is acting it the manner you think"
  "it should. If the pin is right, but the LED doesn't work, but the LED works"
  "when held high, you probably have a bad wire and should replace the connection."

  )
 nil
 nil

(make-node 'nodekitswitchproblem

  "If you've checked the operation of your switches and verified that each works"
  "correctly, then your current problem is probably not with the switch itself."

  "Check the connection you've made to the switch and make sure it goes where you"
  "think it goes and make sure that the connection is good. After that, make"
  "sure that the pin you have it connected to is acting it the manner you think"
  "it should. If the pin is not following the switch, you probably have a bad"
  "wire and should replace the connection between the switch and the pin."

  )
 nil
 nil

)
The HEX LEDs on the Nubus kits are not intuitively obvious in how they function. Nonetheless, complete instructions are given in the Nubus Kit handout that you should have received with your kit (if you didn’t, you can get a copy from the 6.111 locker, or by complaining to a TA).

In short, most problems with the HEX LEDs is because you aren’t using them appropriately. To use them correctly, check that you have done the following:

- Check that you have wired the NUHEX and NUSW to ground. These must be wired low in order for you to (conveniently) use the display.
- Make sure you are clocking the /LHEX and /HHEX signals. They latch the /ADO-/AD31 pins to the HEX display at every rising edge they see. Often it is convenient to wire these to the internal 10MHz clock, however, keep in mind that this is only a good idea if the data you are putting on the /ADO-/AD31 pins is always valid (hint, hint).
- As previously implied, you need to have your data signals wired to an appropriate group of 4 pins from /ADO-/AD31.
- If the HEX display seems to be doing something weird check the last two parts carefully —— make sure that you have your data wired correctly and that the connections are good, and that the latching clock makes sense with the timing of valid data.
- If you deduce that one of the HEX LEDs is broken, you can have it replaced at the desk.

Problems with the interface between the Nubus Kits and the Logic Analyzers are a large and varied lot. Some of the more common ones, however, are addressed here:

1) If you can’t seem to get anything out of the Logic analyzer check and make sure that you are looking for signals on the same POD that you have plugged into your kit. Also make sure that your signals are plugged into the same kit-to-POD interface that matches the interface into which your POD is connected.

2) If you are trying to use the STATE ANALYZER and are getting a ‘Slow or Missing Clock’ error, check to see that you have plugged your system clock into the L#-CLK line at the top of the right hand analyzer bus. Also check that your analyzer is using the correct Master Clock signal.

3) Occasionally parasitic capacitance from the Logic Analyzers will interfere with your digital circuits, causing them to not work when the analyzer is plugged in. Eit. This is rare and if you think this is happening, talk to a TA about it.

Would you like to get some general information about the Nubus Lab kit? [y/n]

The Nubus Kits are a fairly good, very durable, portable lab kit. However,
"there are a few things that you should know about them to avoid killing them."
"In truth, there is little, short of complete annihilation, that you can do to"
"really mess them up as we can replace most parts rather easily.  this is not a"
"a warrant for you to go and try to destroy them, though."

"Some things that you should know to help keep your kit in the finest of working"
"orders:"

"o The wire used to make connections in the kits should be between AWG 26-20."
"the stuff you get from the desk is AWG 22 and that is probably the best"
"size to use --- smaller wire can give a flaky connection and larger wire"
"can damage the spring loaded contacts.  You should also avoid stranded"
"wire as it is difficult to work with (for our purposes)."

"o The digital and analog +5 supplies should NEVER be tied together.  They"
"are not rated for the same output currents and so connecting them, aside"
"from pushing one supply askew, will probably result in blowing the analog"
"supply (or at least its fuse).  SO DON'T DO IT."

"o The various supplies are rated for various amounts of current:"n"The digital +5 supply is rated at 8 Amps."
"The analog +12 and -12 supply are rated at 1.5 Amps."
"The analog +5 supply can supply about 1 Amp; however, keep in"
"mind that it is derived from the +12 supply, so any current"
"draw on the +5 supply counts towards the +12 supply."

"o Finally, any problem with the kit components can be taken care of at the"
"desk. Extra chips can be gotten from the desk and bad chips can be"
"replaced.  Any kit that is suffering some defect can be referred to the"
"desk and it will be fixed.  Occasionally these fixes mean replacing the"
"kit that the student is using, or uprooting anything on the kit; as such,"
"it is advisable to check out your kit parts as soon as you get your kit."


A.3 Tools.1
This file contains the node descriptions of the software section of the RTFM 6.111 Lab expert. It attempts to provide relevant information for all software tools (and the machines they interface) found in the 6.111 lab (and on athena for software).

Tools examined:

Just to clarify how to make a node in the knowledge base:

(make-node "id" "info" "query" "links") is the command to create a node

_id is the name and identifier of the node
_info is a "list" of strings containing information to be printed when the node is activated
_query is a list of strings containing a question. If the node is a dead-end then _query should be "()"
_links is a list of response—node lists. Each response—node list is a list of the form (response node1 node2 ...) where node1, node2, etc are the nodes to be activate if _query is answered with response.

Just a column reference:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
</table>

(make-node 'nodeToolsProblems nil

"We have information on most of the software tools used for 6.111, with which of them are you having a problem:

- [R]educe
- [P]al
- Palas[gn]
- [F]smc
- Fsm[d]
- [A]ssem
- Pro[m]dio

")

(r {nodeReduceproblem})
(p {nodePALprogramming})
(f {nodeFSMCproblem})
(d {nodeFSMDproblem})
(a {nodeAssemproblem})
(g {nodePALASGNproblem})
(m {nodePROMprogramming})

(make-node 'nodeReduceproblem

"Reduce is a fairly straightforward program for simplifying logic equations," and as such, most people find it fairly easy to use and problem free."

"However, two problems do tend to occur fairly often:

- Remember to end your equation lines with a ';'. This is not absolutely critical, but (particularly when using stdin) it will make your output more readable."

- The most common problem is with getting REDUCE to read a file."

"REDUCE is meant to take input from stdin and output to stdout, so you have to use CSH commands in your call to REDUCE to get it to use a file instead. Namely, a '<' indicates the input file and a '>' indicates the output file; so, to run REDUCE with an input file called 'input.in' and an output file called 'output.out' we would invoke REDUCE with:"
% reduce < input.in > output.out

(defun make-node 'nodePALprogramming
  nil
  '(("In what manner are you having problems programming your PAL:
    "
    " Your .pal [f]ile produces an error when read.
    " Your PAL produces an error while [b]urning."
  )
  )(if [nodePALfileerror])
    (b [nodePALburnerror])
  )

(defun make-node 'nodePALfileerror
  nil
  '(("If you are having problems getting the PAL program to read your .pal file"
    "Make sure to check the line that it marks as problematic; PAL is generally"
    "pretty good at identifying where errors are.  PAL is also pretty good at"
    "telling you what is wrong.  As such, which of the following error messages"
    "did it give you:"
    " Can't find spec for given [p]art type"
    " Can't process [a]rchitecture specification"
    " [S]yntax Error"
    " [U]ndefined Symbol"
    " [O]utput must be <foo>'d"
    " [M]ust/may not write equation for complement of pin name"
  )
  )

(defun make-node 'nodePALfilepart
  (("Something is wrong with your device identifier.  PAL doesn't recognize it"
    "for some reason, so check to make sure its correct.  Common errors that"
    "look right are '22v8', or '20v10'."
  )
  (("If your part is correct, and its a nonstandard part, PAL may not have"
    "specifications for it.  If that is the case, talk to a TA about burning your"
    "chip."
  )
  nil
  nil

(defun make-node 'nodePALfilearch
  (("PAL is unable to figure out how you want to configure your output pins."
    "Usually this error is preceded by another identifying message; common"
    "messages and their likely causes are:"
    " Architecture specification too short."
    " You don't have enough output tokens.  Check your part type and your"
    " architecture line to make sure you have enough tokens and that they"
    " match your desired configuration."
  )
  nil
  nil

(defun make-node 'nodePALfilesyntax
One of the more common, and general, messages is the 'syntax error'. This almost seems like PAL's default message, but in truth PAL is fairly specific in applying it. As such check where the error is and then refer to the following general guidelines for help in finding it:

Input Specification:

PAL expects to see at least three (3) lines between architecture declaration and the pin names. If you have too few, then PAL will skip over some or all of your pin names and will become confused when it starts trying to read those pin names.

PAL also uses this descriptor when you have too many pin names. Check that you have the right number of names; common errors are to leave in 'unused' pin names like 'pin5' after you add a name for it. Also, occasionally you see an accidental space in a pin name —— spaces of course make the PAL program think your are naming two pins instead of one.

You also, occasionally, get this error in combination with an 'Undefined Symbol' error. If this is the case, you probably have a typo in an equation line such that a pin name in the equation is not the same as the name actually given to it (check your pin_names lines).

A common typo is to accidentally use a '0' for a '0'.

You also, occasionally, get this error in combination with an 'Undefined Symbol' error. If this is the case, you probably have a typo in an equation line such that a pin name in the equation is not the same as the name actually given to it (check your pin_names lines).

A common typo is to accidentally use a '0' for a '0'.

If PAL tells you 'output must be registered/combinational', then PAL senses a minor discrepancy between your token assignment and your equation format. Registered and combinational outputs use slightly different equal indicators:

A registered output's equation should be written as

<pinnameA> := <sum_of_products>

A combinational output's equation should be written as

<pinnameB> = <sum_of_products>

Note that the registered output uses a ':=', where as a combinational output uses '='.

The three places where you can indicate whether a pin is itself, or its complement. Each location, though has a slightly different meaning. The fact that you got an error indicating that you 'must' or 'may not' write the equation for the complement means that one or more of these locations needs altering.

The three locations are: 1) the architecture line; 2) the pin name; 3) the equation. Each has a slightly different meaning when the complement sign '//' is added to a token or pinname.
"o For the architecture tokens, a '/' indicates that you want the PAL to" "include an inverter between the output gate of the signal and the output pin."

"o The pin name is simply that, a name, thus a '/' indicates you want to" "implement a negative logic signal."

"o Finally, a '/' before an equation indicates that you are realizing the" "complement of the signal with equation in question."

"You need to be careful how you handle these, in order to make sure you realize" "the signal you want, with the equation you have. In general, any combination" "which sums to an even total (0 or 2) of '/'s is a valid combination."

(make-node 'nodePALburnerror
nil

'(The DATA I/O's and the 2900 are generally good machines, though they are often" "cryptic in their error messages. The most common problem involves getting a" "'burnit()' error, coupled (maybe) with an error message number.

"First off, did you receive a 'burnit()' error? [y/n]

'(y {nodePALburnburnit})
(n {nodePALburnother})

(make-node 'nodePALburnother
nil

'(Did the programmer give you an error number? If so enter the number, if not" "enter [n]:

'(n {nodePALburnnothing})

(make-node 'nodePALburnnothing
nil

'(01 {nodePALburn01})
(02 {nodePALburn02})
(1 {nodePALburn01})
(2 {nodePALburn02})
(17 {nodePALburn17})
(1f {nodePALburn1f})
(20 {nodePALburn20})
(21 {nodePALburn21})
(22 {nodePALburn22})
(23 {nodePALburn23})
(24 {nodePALburn24})
(25 {nodePALburn25})
(26 {nodePALburn26})
(27 {nodePALburn27})
(28 {nodePALburn28})
(29 {nodePALburn29})
(2a {nodePALburn2a})
(2b {nodePALburn2b})
(2c {nodePALburn2c})
(2d {nodePALburn2d})
(2e {nodePALburn2e})
(30 {nodePALburn30})
(31 {nodePALburn31})
(32 {nodePALburn32})
(33 {nodePALburn33})
(34 {nodePALburn34})
(35 {nodePALburn35})
(36 {nodePALburn36})
(37 {nodePALburn37})

""
(make-node 'nodePALburnnothing
'"If the programmer crashed without telling you anything something is wrong."
"Find a TA to investigate the problem."
)
nil
nil

(make-node 'nodePALburnburnit
nil
'"Okay, did you receive an error message number? If so enter the number, if not"
"enter [n]: ")
'((n {nodePALburnnonumber})
 (01 {nodePALburn01} {nodePALburnnonumber})
 (1 {nodePALburn01} {nodePALburnnonumber})
 (02 {nodePALburn02} {nodePALburnnonumber})
 (2 {nodePALburn02} {nodePALburnnonumber})
 (17 {nodePALburn17} {nodePALburnnonumber})
 (1f {nodePALburn1f} {nodePALburnnonumber})
 (20 {nodePALburn20} {nodePALburnnonumber})
 (21 {nodePALburn21} {nodePALburnnonumber})
 (22 {nodePALburn22} {nodePALburnnonumber})
 (23 {nodePALburn23} {nodePALburnnonumber})
 (24 {nodePALburn24} {nodePALburnnonumber})
 (25 {nodePALburn25} {nodePALburnnonumber})
 (26 {nodePALburn26} {nodePALburnnonumber})
)
"The burnit() error is a common indicator of a number of problems; in general,
it indicates that the DATA I/O has a problem talking to the PAL. This is
often because of an invalid Family/Pinout code. As such, recheck that your
code is the right one for your part."

"Burnit() errors can also occur because of a damaged chip. Check that the chip
is in good order, particularly that all pins exist and are in the correct
socket of the programmer."

"Finally, burnit() errors occasionally occur because some internal mechanism"
"of the programmer cuts out. If you've checked everything else, try power"
"cycling the programmer to reset it. BE CAREFUL --- LET THE PROGRAMMER REST"
"A COUPLE SECONDS AFTER POWERING DOWN before you turn it back on. After it"
"finishes its self test, select the RS232 PORT selection with the arrow keys,"
"and hit 'enter'; 'Computer Control' ought to appear under 'RS232 Port', hit"
"'enter' again so that only 'Computer Control' is showing. (Note, these"
"instructions are for the DATA I/O programmers; if you are using the 2900"
"follow the reboot instructions on the machine). Now, trying burning your PAL"
"again."

"You may also try burning your chip on another programmer, and you should"
"certainly try this if you get another burnit() error after cycling the"
"programmer."

"If all else fails, try programming a different chip; it is possible that your"
"chip is fried."

"If that doesn't work, find a TA to investigate the problem."

;;; The following nodes (nodePALburn#) contain information about the error messages
;;; from the DAT I/O 200 and 2900 programmers. Much of the information and description
;;; was lifted (some almost directly) from the manuals for those machines.

(make-node 'nodePALburn01

'("01: EPROM FAIL"
" This is a self-test error message, and it indicates that the EPROM"
" configuration is wrong. Get a TA to investigate."

))

(make-node 'nodePALburn02

'("02: SRAM FAIL"
" This is a self-test error message which indicates that the 212 programmer"
" has detected a verification error in its SRAM. Get A TA to investigate."

))

(make-node 'nodePALburn17

'("17: BANK ERROR"
" This message indicates that the bank selector of the 212 programmer is unable"
" to switch banks. This is a hardware problem, so go get a TA."

))

(make-node 'nodePALburn1f

'("1f: CANNOT ERASE DEVICE ERROR"
" This message is given if the 2900 programmer was unable to erase an"
" EEPROM. This usually occurs because of a dead chip --- try another"
" one; if you get the same error, get a TA."

))

(make-node 'nodePALburn20

'("20: NON-BLANK DEVICE"
" This message is given if the programmer has performed a blank check"
on a device and detects bits that are not in their blank or erased state,"
"and are not illegal bits."
)
nil
nil

(make-node 'nodePALburn21
'("21:  ILLEGAL BIT ERROR"
"This message appears when the programmer detects a bit programmed to"
"the incorrect state within a chip. When this error occurs, try reprogramming"
"the device (erasing it first if necessary); continuing errors are likely"
"the result of a defective chip."
)
nil
nil

(make-node 'nodePALburn22
'("22:  DEVICE PROGRAMMING ERROR"
"This message indicates that the programmer has detected a defective"
"memory cell while programming a chip. The chip is bad, try another one."
)
nil
nil

(make-node 'nodePALburn23
'("23:  VERIFY DATA ERROR"
"This error indicates that the programmer has found a memory cell which"
"was not programmed correctly. Try reprogramming and/or another chip."
)
nil
nil

(make-node 'nodePALburn24
'("24:  VERIFY DATA ERROR"
"This error indicates that the 2900 programmer has found a memory cell which"
"was not programmed correctly. Try reprogramming the chip. If the problem"
"persists, try another chip."
)
nil
nil

(make-node 'nodePALburn25
'("25:  BAD DEVICE CODE"
"This message indicates that an invalid family/pinout code was entered to"
"the 212 programmer."
)
nil
nil

(make-node 'nodePALburn26
'("26:  WRONG SOCKET"
"This message indicates that you have inserted your chip into the wrong"
"socket of the 212 programmer. Switch sockets and try again."
)
nil
nil

(make-node 'nodePALburn27
'("27:  END OF USER RAM EXCEEDED"
"There is not enough user RAM for the amount of data you want to load into it."
"Your device block size may be too large, or your beginning address to high."
In any case, the operation may still be performed, but only part of the data will be programmed.

(make-node 'nodePALburn28
('("28: FATAL DEVICE-SPECIFIC PROGRAMMING ERROR"
  "A generic error code generated by the device-specific algorithm of the 2900 programmer. However, the error is serious enough to halt the operation being performed."
))

(make-node 'nodePALburn29
('("29: NON-FATAL DEVICE-SPECIFIC PROGRAMMING ERROR"
  "This is a warning message only. It is for informational purposes only and is therefore not fatal to the operation being performed, which in most cases will be completed successfully."
))

(make-node 'nodePALburn2a
('("2A: DEVICE INSERTION ERROR"
  "This message is given when one of the following problems occurs on the 2900:
  o The chip socket is not in the lock position.
  o The chip is inserted backwards.
  o The chip is not bottom justified in the socket.
  o The pins of the chip are not making good contact.
  "
  Check the positioning of the chip in the socket and try programming it again.
  If the error persists, try a different chip."
))

(make-node 'nodePALburn2b
('("2B: STRUCTURED TEST ERROR"
  "The 2900 programmer has performed a function test on a chip and detected a failure. It is likely that this is caused by a faulty chip."
))

(make-node 'nodePALburn2c
('("2C: STRUCTURED TEST ERROR"
  "The 2900 programmer has performed a function test on a chip and detected a failure. It is likely that this is caused by a faulty chip."
))

(make-node 'nodePALburn2d
('("2D: BASE FOR DEVICE NOT INSTALLED"
  "The chip you selected cannot be programmed in the Base that is presently installed. No device operations may be performed for your chip until the correct Base has been installed."
))
"2E: PROGRAMMING HARDWARE HAS NOT PASSED SELF-TEST"
" This message occurs when an operation is attempted, but the programmer (2900)
" has failed the self-test for some of the necessary hardware. Reboot the
" 2900 programmer and check the self-test screen. Get a TA to investigate."

"30: DEVICE ALGORITHM NOT FOUND"
" This message will be given if the family/pinout codes selected do not
" correspond to an algorithm supported by the 2900 programmer."

"31: DEVICE OVER-CURRENT FAULT"
" This message indicates that the socketed chip has a higher programming
" current than the chip you identified it as (i.e. than it is supposed to)."
" The device may be faulty: try reprogramming and then, if that fails, another
" chip."

"32: Vcc ERROR"
" This message indicates that when +5 Volts is applied to your chip, the voltage
" drops to an unacceptable level. Your chip is likely faulty, so try another.

"33: Vpp ERROR"
" The power supplies of the 212 programmer cannot be set to the proper levels
" for your chip. Check your chip type and verify with a TA that your procedure
" is correct.

"34: BAD DEVICE CODE"
" An incorrect family/pinout code was given to the 212 programmer."
" Recheck your code and try again. If the error persists try another chip.

"35: NOT TRISTATE"
" The 212 programmer was unable to tristate all pins as specified. Your chip
" is likely faulty, try another."
(make-node 'nodePALburn36
'("36: DEVICE NOT ENABLED"
" Not all of your chips data pins are being driven when the chip is activated."
" Your chip is probably bad, and you should try programming another."
)
nil
nil
)

(make-node 'nodePALburn37
'("37: BAD INSERTION"
" The chip was not inserted into the 212 programmer correctly. Be sure that"
" the bottommost pin of the chip is in the bottom slot of the 212. The topmost"
" chip pin should be in the top slot. Be sure that pin 1 is one of the topmost"
" pins."
)
nil
nil
)

(make-node 'nodePALburn39
'("39: SYNCHRONOUS ENABLE ERROR"
" The chip being verified was programmed with synchronous programming."n"
" enabled, therefore, sync. programming must be enabled when verifying."
" Make sure sync. programming is enabled and reverify the chip."
)
nil
nil
)

(make-node 'nodePALburn3a
'("3a: SYNCHRONOUS DISABLE ERROR"
" The chip being verified was programmed with synchronous programming disabled."
" Synchronous programming must be disable to verify your chip, check that"
" sync. programming has been disabled and try to reverify your chip."
)
nil
nil
)

(make-node 'nodePALburn3b
'("3b: NO DEVICE PRESENT"
" This message indicates that the programmer (2900) does not detect a chip"
" is the programming socket."
)
nil
nil
)

(make-node 'nodePALburn40
'("40: I/O INITIALIZATION ERROR"
" This message is given after an attempt to initialize the Remote port has"
" failed. Have a TA check connections between the programmer (2900) and the"
" host and try the operation again."
)
nil
nil
nil
)

(make-node 'nodePALburn41
'("41: SERIAL-FRAMING ERROR"
" The Remote port has detected a start bit, but the corresponding stop bit"
" is incorrectly positioned. Have a TA check the baud rate and the stop bit"
" setting for the connection."
)
nil
)
(make-node 'nodePALburn42
  '("42:  SERIAL-OVERRUN ERROR"
    " The Remote port has received characters that the programmer was"
    " unable to service. Have a TA check the baud rate and stop bit settings for"
    " the Remote port."
  )
)

(make-node 'nodePALburn43
  '("43:  SERIAL FRAMING/OVERRUN ERROR"
    " The Remote port has detected a combination of improper start/stop bit"
    " placement and has received characters that the 2900 programmer was unable"
    " to service. Have a TA check the baud rate and stop bits of the port."
  )
)

(make-node 'nodePALburn46
  '("46:  I/O TIMEOUT"
    " The programmer did not receive a data file during download operation."
    " Have a TA investigate the problem."
  )
)

(make-node 'nodePALburn48
  '("48:  BUFFER FULL"
    " The serial port input buffer of the 212 programmer received more characters"
    " than it was able to process. Get a TA to investigate, specifically, to make"
    " sure that the handshake lines are hooked up and working correctly."
  )
)

(make-node 'nodePALburn52
  '("52:  DATA VERIFY ERROR"
    " The data from the Remote port did not match the data in RAM, thus"
    " verification failed. Check the data and try again."
  )
)

(make-node 'nodePALburn63
  '("63:  DRAM ADDR# FAIL"
    " This message is a self-test error message indicating that the DRAM"
    " of the 212 programmer was found to be faulty. Notify a TA of the problem"
    " and try another programmer."
  )
)

(make-node 'nodePALburn64
  '("64:  DRAM # ERROR"
    " This message is a self-test error message indicating a DRAM failed a"
    " verification check in the 212 programmer. Notify a TA of the problem and"
    " try doing your work on another programmer."
  )
)
(make-node 'nodePALburn76
  '("76: PIN 23 ERROR"
    " The self-test of the 212 programmer found an inaccurate voltage level on"
    " socket pin 23. Get a TA to investigate."
  )
  nil
)

(make-node 'nodePALburn75
  '("75: SECURITY FUSE VIOLATION (2900) or"
    "75: PIN 22 ERROR (212)"
    " This message, from the 2900, indicates that the security fuse of your chip"
    " has been programmed. Use a master device whose fuse is still intact."
    ""
    " From the 212, this message indicates that the self-test found an inaccurate"
    " voltage level on socket pin 22. Get a TA to investigate."
  )
  nil
  nil
)

(make-node 'nodePALburn74
  '("74: DATA BUS FAIL"
    " This error occurs when either your chip has a faulty data bus or the ZIF"
    " socket of the 212 programmer is faulty. Try programming another chip; if the"
    " problem persists, notify a TA and switch to another programmer."
  )
  nil
  nil
)

(make-node 'nodePALburn73
  '("73: DATA SHORT"
    " Your chip is faulty, it contains a short on one of its data lines. Try"
    " programming another chip."
  )
  nil
  nil
)

(make-node 'nodePALburn72
  '("72: ADDRESS SHORT"
    " Your chip contains a short on one of its address lines. As such, your chip"
    " is bad and you should replace it and try programming again."
  )
  nil
  nil
)

(make-node 'nodePALburn71
  '("71: PIN 1 ERROR"
    " This message indicates that the self-test found the pin 1 socket to be at"
    " an inaccurate level. Get a TA to investigate the problem."
  )
  nil
  nil
)

(make-node 'nodePALburn67
  '("67: I/O FORMAT ERROR"
    " This message indicates that the data sent to the 212 programmer was not in"
    " the expected hexadecimal format. Check the I/O format for your data."
  )
  nil
  nil
)

nil
)
This message indicates that the 2900 programmer cannot program the security fuse in your chip; as such your chip may be defective and you should try using another chip.

Coming from a 212 programmer, this message indicates that the self-test has found an inaccurate voltage level on socket pin 26. Get a TA to investigate.

This message indicates that the 2900 programmer cannot program the security fuse in your chip; as such your chip may be defective and you should try using another chip.

Coming from a 212 programmer, this message indicates that the self-test has found an inaccurate voltage level on socket pin 26. Get a TA to investigate.

The self-test of the 212 programmer found an inaccurate voltage level on socket pin 26. Get a TA to investigate.

This message, on the 2900, indicates that a preload vector in the programming data cannot be applied to the logic device.

From the 212, it indicates that the self-test has found a faulty voltage level on socket pin 28. Get a TA to investigate.

The Remote port detected data with an incorrect parity; check the setting on the port.

The sumchecks did not match between the host computer (before download) and the programmer after download. Try the operation again, if the problem persists, get a TA to investigate.

This message indicates that the 212 programmer has determined in its self-test that an internal part has failed. Notify a TA.

This message indicates that the 212 programmer has determined in its self-test that an internal part has failed. Notify a TA.
"84: I/O FORMAT ERROR (2900) or"  
"84: 8268 FAILURE (212)"
" There is a compatibility problem between your data translation format and"
" that expected by the 2900 programmer. Refer to the programmer's manual"
" and get a TA to investigate."
" With the 212 programmer, this message indicates that the self-test has"
" determined that an internal part has failed. Notify a TA."

If the 212 programmer, this message indicates that the self-test has"
" determined that an internal part has failed. Notify a TA."

"86: INVALID NUMBER OF PARAMETERS (2900) or"
"86: MANUFACTURER ID ERROR (212)"
" 2900: A CRC command was issued with the improper number of parameters"
" preceding it."
" 212: The family/pinout code is incorrect. Verify your code and try again."

"89: ILLEGAL PARAMETER VALUE (2900) or"
"89: INCORRECT DEVICE (212)"
" 2900: A CRC command was issued with an improper parameter."
" 212: You have attempted to perform an operation which is not supported"
" for your current chip. You should either change chip types or"
" request a different operation."

"8a: CAN NOT ERASE"
" The 212 programmer cannot erase the PLD selected. Use a different type"
" of chip."

"8B: ERROR RESTORING/SAVING CRC USER-DEFINED PARAMETERS OR DEFAULTS (2900) or"
"8B: ILLEGAL OPERATION (212)"
2900: An error occurred while attempting to save/restore parameters. Get a TA to investigate the problem.

212: You have attempted to perform an operation which is not supported for your current chip. You should either change chip types or request a different operation.

212: You have attempted to program a Lattice chip that has been programmed as a master chip. The 212 will not do this. You must use either a blank chip or unprogram the master fuse.

8d: XMIT SUMCHECK ERROR
The 212 programmer has detected a discrepancy between the calculated sumcheck of the sent data and the sumcheck received. Try programming your chip again. If the problem persists, have a TA check the connection between computer and programmer.

8E: FILE ERROR (2900) or NOT LATTICE DEVICE (212)
2900: A disk file error occurred while the 2900 was accessing a disk. Check that, if you are saving something, the disk is not write-protected.
212: The requested command can only be used with a Lattice chip. Choose another operation, or use a Lattice chip.

8F: NON-JEDEC DATA PRESENT IN FILE (2900) or USER RAM INCOMPATIBLE DEVICE (212)
2900: Some data in your file is not of the proper format.
212: You attempted to program or verify a chip with incompatible fuse data. Check your data and try again.

90: ILLEGAL I/O FORMAT
You tried to select an I/O format that is not supported by the programmer.

91: I/O FORMAT ERROR
This message indicates that the 212 programmer received an invalid
"98: END OF DEVICE EXCEEDED (2900) or"
"98: DEVICE CHECK FAILURE (212)"
  2900: There is not enough user RAM for the amount of data you want to load into it. Your device block size may be too large, or your beginning address to high. In any case, the operation may still be performed, but only part of the data will be programmed.

  212: Your chip has failed a device check. This may be caused by improper insertion into the 212 programmer, or an invalid family/pinout code. It is also possible that your security bit has been programmed. Verify your family/pinout code and check the positioning of the chip and try reprogramming. If the problem persists, try a different chip.

"99: END OF FILE EXCEEDED"
The memory block size and the begin address you specified are too large for the data file you intend to use. Check the specified parameters and modify them for your data.

"9A: ALGORITHM DISK CANNOT BE FOUND"
This message is given if you are selecting a chip algorithm and the system disk of the 2900 is not in the drive.

"9D: I/O ADDRESS BEYOND RANGE OF DATA FORMAT SELECTED"
You ran out of space. An I/O address in your data exceeded the maximum address of your chip. Check your data format and the block size and beginning memory address.

"A1: NO ELECTRONIC ID"
Your chip does not have an electronic ID. This could be caused by an invalid family/pinout code.

"A2: ELECTRONIC ID VERIFY ERROR"
The chip you tried to programmer did not have the correct ID. This could indicate an incorrect family/pinout code. Verify your code and try again.

"A3: VPA SUPPLY FAILURE"
The self-test has indicated that the programming power supplies cannot be set at the proper levels. Notify a TA and try a different programmer.
(make-node 'nodePALburna4
  "A4: VSB SUPPLY FAILURE"
  "The self-test has indicated that the programming power supplies cannot"
  "be set at the proper levels. Notify a TA and try a different programmer."
  )

(maknode 'nodePALburna5-f
  "A#: SWITCH FAILURE (212)"
  "The self-test indicates that an internal switch circuit has failed."
  "Notify a TA and try another programmer."
  )

(maknode 'nodePALburnab
  "AB: UNABLE TO LOAD SYSTEM FILE FROM SYSTEM DISK (2900)"
  "You attempted to exit or suspend CRC mode (2900) when the system disk"
  "was not in the drive. Insert the disk and continue."
  )

;;;; This is an interesting one... they seem to know what will exist in the future...

(maknode 'nodePALburnac
  "AC: SECURITY VIOLATION (2900)"
  "You attempted to use a new version of software that has not been installed"
  "yet. Exit CRC mode and use the Update command to install the new software."
  )

(maknode 'nodePALburnb2
  "B2: PARTIAL DEVICE OPERATION NOT ALLOWED"
  "Your chip only supports the default block size, change your block limits"
  "and try again."
  )

(maknode 'nodePALburnb4
  "B4: ODD MEMORY BEGIN ADDRESS NOT ALLOWED"
  "A chip of >= 16 bits can not have an odd memory begin address. Change your"
  "parameter and reprogram the chip."
  )

(maknode 'nodePALburnc3
  "C3: SECURITY IS PROGRAMMED"
  "The 212 has detected that the security bit of your chip has been programmed."
  "The selected operation cannot be performed while this bit is programmed;"
  "thus you should use a blank device or unprogram the security bit."
  )
nil

(make-node 'nodePALburn0-f

'(D#: DATA BUS FAILURE (212))
"This message from the 212 indicates that the self-test has detected a"
"failure in the data bus. Notify a TA about the problem and try another"
) nil nil nil

(make-node 'nodePALburnd5

'(D5: PORT TRANSFER ERROR (2900))
"Your serial port is not properly connected."
) nil nil nil

(make-node 'nodePALburne0-f5

'(E or F#: ADDRESS BUS FAILURE"
"Indicates that a failure has been detected in the address bus. Notify a TA"
"about the problem and try another programmer."
) nil nil nil

(make-node 'nodePALburnf6

'(F6: VCT1 FAILURE"
"The 212 has detected that the VCT1 supply is not working. Notify a TA and"
"try another programmer."
) nil nil nil

(make-node 'nodePALburnfe

'(FE: UNDEFINED ERROR"
"An error occurred that the programmer (2900) could not categorize. Get"
"a TA to try to help sort things out."
) nil nil nil

;;; The following section deals with the FSMC program which uses a C-like language
;;; to construct .pal files for finite state machines.

(make-node 'nodeFSMCproblem

'(FSMC is a complicated program and it is HIGHLY recommended that you read the"
"section of the FSM Design Tools Handout that deals with FSMC. Doing so will"
"greatly decrease the number of errors you have in your .fsm files."
""
"Also, note which line FSMC indicates has a problem. Most problems in FSMC"
"are attributed to the right line, so as you learn more about the problem,"n"
"look for the error there first."
"
"Select the error message that FSMC is giving you:"
"[M]issing operator"
"[L]abel: 'foo' referenced but never defined"
"[I]mproper statement termination"
" [0]ther"
)
'((m {nodeFSMCmissing})
(l {nodeFSMClabel} {nodeFSMCmissing})
(i {nodeFSMCimproper})
(o {nodeFSMCother})
)
)

(make-node 'nodeFSMCmissing
'("The 'Missing Operator' error message is FSMC's way of telling you that it has"
"encountered something, usually a character, that it doesn't know what to do"
"with.  This is most often done on state or other variable names; it should"
"be noted that FSMC requires variables to begin with an ALPHABETICAL character"
"or a '_'.  Numbers are allowed in variable names, but they cannot begin"
"the name.  Also, remember to place a ';' after a state label."
"
"'Missing operator' could also mean that you are missing a ';' on the previous"
"line.  FSMC depends upon seeing a ';' to know when equations end, and when it"
"finds the beginning of a line before the end of the logic equation is seen, it,"
"understandably, gets a little confused."
)
nil
nil
)

(make-node 'nodeFSMClabel
'("You have referenced a label (a state name) somewhere, but for some reason"
"FSMC never sees the state label defined.  This problem usually occurs in"
"conjunction with another error, often a Missing Operator error.  Check"
"where 'foo' is defined, there is probably an extra or illegal character"
"which is preventing FSMC from recognizing 'foo'."
)
nil
nil
)

(make-node 'nodeFSMCimproper
'("The ';' at the end of a line is important to FSMC to tell it when one line"
"ends and the next is about to begin.  This error message indicates that"
"you have forgotten a ';', probably on the line just before the marked line,"n"
"and FSMC wants to point that out to you.  This message usually occurs when the"
"line without a ';' is a statement line (like 'if') --- if the ';' where missing"
"from an output declaration, you would probably see a 'Missing operator' error."
)
nil
nil
)

(make-node 'nodeFSMCother
'("If you have a problem with FSMC which is not discussed above, read"
"the handout on the FSMC program before asking for help from a TA."
)
nil
nil
)

;;; The following section deals with the FSMD graphical FSMC interface.

(make-node 'nodeFSMDproblem
nil
'("FSMD is pretty much just a graphical version of FSMC."
"
"Problems with FSMD generally fall into two categories --- the first being"
"a problem getting FSMD to run, the second type being problems within the"
"program.  As such, which do you fall under?  Getting it to [r]un, or "

106
The main problem encountered with running FSMD is with an environment variable for the program, and can be identified by the error message:

"Error: Widget drawarea has zero width and/or height"

which appears after you try to invoke FSMD. This problem is easily fixed if you gain access to FSMD by using 'setup 6.111' instead of 'add 6.111'.

"The proper value of the environment variable is set when setup is called."

"For those who really want to use 'add', you can also fix the variable by sourcing the .attachrc in the 6.11 locker before invoking FSMD."

"Thus, to fix the problem, you should do one of the following at your athena prompt:

'.setup 6.111'
  This will give you a new xterm window, from which FSMD will run.

'-or-

'% source /mit/6.111/.attachrc'
  This will add the right environment variables so you can run FSMD from your current window."

The graphical interface of FSMD makes it much easier to conceptualize the development of finite state machines. However, the error messages of FSMD are very cryptic. This is because to generate the error messages (note, that almost all error messages are encountered when 'check'ing the fsm, under the 'Options' menu), FSMD simply creates a temporary file and runs FSMC on it, and gives you back the error messages it produces. This is "exceptionally difficult to understand as the errors are listed by line number, and yet, within FSMD, no line numbers exist."

"The easiest way to figure out what is wrong is go into the '/tmp' directory and look at the temporary file. In this manner you nail down exactly what area of your FSM is generating the problem and you can simply refer to the FSMC part of this system."

"In order to access the right temporary file, you may note that the file name is printed before each error message returned through the 'Check FSM' Option."

"Would you like to go to the FSMC section? [y/n]"

Would you like to go to the FSMC section? [y/n]

Which of the following error messages most closely matches your message or problem:

"Trying [H]ueristic 3..."
"Warning: foo cannot be [a]ssigned as specified"
"Cannot allocate [p]in for <bartype> output foo, abort"

Would you like to go to the FSMC section? [y/n]
Heuristic 3 is Palasgn's last ditch effort to get it all in there. In general, any output from Heuristic 3 will have some errors. Sometimes, there will be "no real output with the .pal files being blank; sometimes a few signals" have been arbitrarily left out; "occasionally everything will be fine." Generally, if you hit Heuristic 3, check over the PAL files _very_ carefully before using them.

"Now, why might it have to resort to H3? Usually, H3 is encountered because you are trying to jam too much information on too few PALs. Count your number of outputs you have, and make sure your PALs have at least that many" "output pins (always necessary). Also, check your pin equations and your device type. Most PALs have a limited number of summation terms that they can handle. The 20v8, for example can't handle anything over 8 terms on any" "of its pins. The 22v10 fares better (take a look at its data spec) and can handle more terms on some of its pins. Lack of output space, whether for raw output numbers, or for pins that can handle the equations, is the "primary cause for reaching, and failing, Heuristic 3."

This message indicates that Palasgn cannot assign a signal as specified in your file. This is a non-fatal error, but it does mean that your pinout will look slightly different than you expect. You may wish to check the .eqn files or change the type of PAL you use to give Palasgn more room to work; a few small changes may allow you to return to your preferred pinout without any problems.

Palasgn cannot allocate a signal that you need to _any_ pin in the PAL. This is, of course, a problem, and one that often enacts successive heuristics, "and often results in failing to deliver a usable .pal file."

"Basically, you are trying to jam too much information in too small a place." Count the number of outputs you have, and make sure your PAL(s) have at least" "that many output pins (always necessary). Also, check your pin equations and" "your device type. Most PALs have a limited number of summation terms that" "can handle. The 20v8, for example can't handle anything over 8 terms on" "any of its pins. The 22v10 fares better (take a look at its data spec) and" "can handle more terms on some of its pins. Lack of output space, whether for raw output numbers, or for pins that can handle the equations, will often result in a signal not being allocatable, so check these areas for problems in your design."

"In what manner are you having problems programming your PROM:" "Your .ntl [f]ile produces an error when read."
Your PROM produces an error while burning.

((f {nodePROMfileerror})
  ((b {nodePROMburnerror} {nodePALburnerror}))
)

(make-node 'nodePROMfileerror
  (((c "If you are having a problem with PROMDIO reading a file, make sure that that"
      "file is in the proper format --- namely, that the file is a *.ntl file."
    )
    "If it is not, then you need to convert it. Odds are that it is a *.dat"
    "file, and all you need to do is run 'dat2ntl' on it."
    )
    "If it is a *.ntl file, then something has corrupted your file and you should"
    "try to remake the file using whatever programs you previously used (most likely"
    "Assem and your .as and .sp files; or mtable/mulprom and dat2ntl). If the"
    "problem persists, check the man pages for the programs you are using and ask"
    "a TA for help."
  )
)

(make-node 'nodePROMburnerror
  (((c "The errors generated by PROMDIO while burning a chip are nearly identical"
      "to those of PAL, as such you are going to be shunted to the PAL burn error"
      "section. If in this section RTFM mentions the PAL program, treat it as if"
      "it is talking about PROMDIO."
    )
    (nil nil)
  )
)

(make-node 'nodeAssemproblem
  (nil)
  (((c "A common problem with the Assem programs is that they often seem absent when"
      "students wish to use them. Namely, you try to use them and athena tells you"
      "something along the lines of, 'What?' To fix this problem it must be"
      "understood that the majority of the Assem 'programs' are just scripts which"
      "refer to another program 'assembler'. Consequently, if you have attached"
      "the 6.111 locker using the 'attach' or 'add' commands, the /scripts directory"
      "in the 6.111 locker has not been added to your path. To fix this, you should"
      "do one of the following at your athena prompt:"))
  (nil nil)
)

"% setup 6.111"
"This will give you another xterm which will have the right path information."
"-or-"
"% source /mit/6.111/.attachrc"
"This will source the applicable information and add the right information"
"to your path."

"Now, given that you are actually making a successful call to Assem, a problem"
"is likely to involve Assem ceasing its processing of your file and telling"
"you something is wrong. Also, note that Assem will try to tell you what is"
"wrong, as such, much of this section is redundant to what you'll have already"
"seen from Assem. Now, which of the following matches the descriptor you"
"received from Assem:
" error [6]: could not open specified spec_file, 'foo.sp'
" warning, [c]ommand fields overlap when adding '<command>'
" [o]ther (perhaps, even no error message at all)"
" error [14]: 'foo' found in code but not defined or found as a label name"
Your error is likely to have been caused by the absence of a colon after a label. Assembler is complaining that it found something that doesn't look like a label (because it has no colon), and yet it is not described in the specification file as anything else.

In general this message indicates some character string is unrecognized. It may be a label that's missing a colon, a comment not separated out, or even a misspelled command (like 'jump' instead of 'jmp').

It indicates has an error and look carefully at it, you should be able to find the problem.

The 'command fields overlap' error is often indicative of a missing semicolon. The error occurs when Assembler detects that it is trying to set the command bits for an instruction for a second time. As two commands can not be executed at the same time, it deduces that you've forgotten a semicolon at the end of a line.

This can also occur when a character is out of place --- a comment indicator or even a 'misspelled' command. Check the line carefully for mistakes, you ought to be able to find the problem.

Check the line number Assem gave you and look for the problem there.

It seems that Assembler couldn't find or read the specification file you list in your .as file. Check that your SPEC_FILE line is appropriate for your specification file.

"Other" type bugs are often some of the most difficult to diagnose, usually because a mistake has been made that the programs won't catch. For example, if your MASK_COUNT is wrong, the Assem scripts will work just fine, they'll just produce an output that you weren't expecting (and will probably result in a burn error of somewsort). As such, these errors often have something to do with the commands _to_ the assembler, as opposed to those to the PROM, so check your assembler commands in your .as file carefully.

If, after looking at the .as file, you still are unsure as to what is happening, ask a TA for help.
A.4 System.l

; File: system.l
; Description: Knowledge base for the RTFM 6.111 Lab expert
; Expansion: KCL version
; Author: Owen Wessling
; Last Mod: 10/11/95 v1.0

;; This file is a part of the RTFM 6.111 Lab Expert project.
;; This is a part of the M.Eng thesis by
;; Owen Wessling, M.Eng EECS '96, MIT.
;; Copyright (c) 1995 Owen Wessling All Rights Reserved.
;; This software is supplied "as is," without any kind of
;; warranty whatsoever, but not limited to the implied
;; warranties of merchantability and fitness for a particular
;; purpose. The author and MIT make no representations about the
;; suitability of this software for any purposes and do not promise
;; to support it in any way.
;;
;; This software is not to be incorporated into a commercial
;; product, or access to it sold, without written permission
;; from MIT. Otherwise, copies which include this notice may
;; be freely made and distributed.

;; This file contains the system debugging techniques nodes for the
;; RTFM 6.111 lab expert system.

;; Just to clarify how to make a node in the knowledge base:

;; (make-node _id_info _query_links) is the command to create a node
;;
;; _id is the name and identifier of the node
;; _info is a "list" of strings containing information to be printed
;; when the node is activated
;; _query is a list of strings containing a question. if the node is a dead-end
;; then _query should be "()"
;; _links is a list of response--node lists. Each response--node list
;; is a list of the form (response node1 node2 ... ) where
;; node1, node2, etc are the nodes to be activate if _query
;; is answered with response.

;; Just a column reference:

(make-node 'nodesystemproblems
nil
'("Many kinds of digital system have particular techniques that can assist in"
  "debugging them. As such, which of the following types of systems or"
  "implementations are you experiencing difficulties with:"
  "[F]SMs"
  "[M]icroprogramming"
)
'(f {nodedefheuristic} {nodesystemFSM})
(m {nodedefheuristic} {nodesystemmicro})

111
Control structures using FSM's are easiest to debug by analyzing the inputs and state of the structure. As such, wire up the state bits of your FSM as well as all of the inputs to the logic analyzer and look to see what it is doing. Check what state the FSM is in, and check what inputs it sees and examine how it reacts. Verify that this is how it should react according to the FSMC files and .pal files you have. When something goes wrong, examine what the FSM was doing and what it saw, then analyze what it did do. Try and figure out what it's doing and why.

The best way to debug a microcoded system is to wire everything up to the logic analyzer and see what's going on. The principle things you absolutely need to look at are the instruction address as it comes out of the sequencer and the instruction codes as they come out of the PROMs. This information tells you the microcontroller is doing and what it thinks it is suppose to do next.

On top of those signals, the condition code bits are useful to have at hand, and it is not a bad idea to check the instruction going back into the sequencer and the instruction words coming out of the registers after the PROMs. With all of this information, you ought to be able to determine what instruction the microcontroller thinks it is on, what it thinks that instruction is, and what it thinks it is suppose to do next. If you examine a problem in this manner you ought to be able to determine what the microcontroller is doing and to isolate the problem.

Debugging systems is hard work and very time consuming. However, the task is often increased by attempting to find 'shortcuts' in debugging techniques, or by using poor debugging techniques. The first thing to understand to make the process shorter is to realize that the Logic Analyzer is your friend. True, it takes a lot less time to plug your logic probe into your kit than it does to plug signals into the Logic Analyzer, but the extra time is WELL WORTH IT. A few bugs, usually the kind that result from wiring mistakes, can be found quickly using a probe. But, even in the cases where a probe will indicate that a problem exists, the logic analyzer will illuminate the problem in far greater detail, allowing for a much better understanding of what is going on. And, as well, the analyzer is capable of locating problems that the probe simply cannot recognize.

As such, get to know the analyzer. Make it your friend. Learn how to use it efficiently and you will vastly decrease the amount of time you spend in lab debugging your kits.
A.5 Design.1

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; File: design.l
; Description: Knowledge base for the RTFM 6.111 Lab expert
; Expansion: KCL version
; Author: Owen Wessling
; Last Mod: 10/10/95 v1.0
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

%%% This file is a part of the RTFM 6.111 Lab Expert project.
%%% This is a part of the M.Eng thesis by
%%% Owen Wessling, M.Eng EECS '96, MIT.
%%% Copyright (c) 1995 Owen Wessling All Rights Reserved.

%%% This software is supplied "as is," without any kind of
%%% warranty whatsoever, but not limited to the implied
%%% warranties of merchantability and fitness for a particular
%%% purpose. The author and MIT make no representations about the
%%% suitability of this software for any purposes and do not promise
%%% to support it in any way.

%%% This software is not to be incorporated into a commercial
%%% product, or access to it sold, without written permission
%%% from MIT. Otherwise, copies which include this notice may
%%% be freely made and distributed.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

%%% This file contains the node descriptions of the design/hardware section
%%% of the RTFM 6.111 Lab expert. It attempts to provide relevant
%%% information for particular problems dealing with design and lab specific
%%% problems likely to be found 6.111.

%%% Just to clarify how to make a node in the knowledge base:

%%% (make-node _id _info _query _links) is the command to create a node

%%% _id is the name and identifier of the node
%%% _info is a "list" of strings containing information to be printed
%%% _query is a list of strings containing a question. if the node is a dead-end
%%% then _query should be "("
%%% _links is a list of response—node lists. Each response—node list
%%% is a list of the form (response node1 node2 ... ) where
%%% node1, node2, etc are the nodes to be activate if _query
%%% is answered with response.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

%%% Just a column reference:

;;;;:0--------1----------2----------3----------4----------5----------6

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(make-node 'nodedesignproblems
nil
'("Design problems are very hard to figure out without a good understanding of"
"what is going on in your system. As such, this section of RTFM will attempt to"
"help you with some of the concepts involved in many design issues, particularly"
"those involving 6.111 labs. When using this section, you should read the"
"given information and then think about what it means. After you use this"
"and, have thought about your system and tried to figure out and test for all"
"the ways something might be going wrong, then go get a TA and ask for help."
""
"Now, to help narrow the field of design, what do you need design help for:"
" Lab [1]"
" Lab [2]"

113
The design issues in Lab 1 are not really circuit based as all the logic can be implemented inside a PAL. However, that does leave the question of "How do you get the PAL equations?" To answer that you should, of course, use a K-map. For each of the outputs you wish to realize, create a truth table which lists all inputs (remember that flip-flops have their previous state as an "input") and all outputs. Turn this into a Karnaugh map and then find the MSP for each output. Voila, you've just realized the PAL equations.

Now, if you're unsure how to put these into a .pal file, read the document "PAL Programming", which will tell you everything you need to know (and then some) about how to create a viable .pal file.

Lab 2 has a number of design issues centering on signal control. Particularly, control over asynchronous input signals and control of output signals. As well, there is the issue of how one goes about designing an FSM.

The following list comprises the main design areas of Lab 2, please indicate in which area you are having problems: 

Handling the input signals (GO, ST, & WR)
Figuring out the system timing (clocks)
Designing the Expired signal counter
Designing the FSM
Something other than the above.

The three primary input signals in Lab 2 need to be conditioned in some manner before given to your FSM. In the simplest case (ST) this simply means that the signal needs to be synchronized with the system clock. The other two however, require a little more work.

The GO signal is the message to the system which tells it to initiate a process. This allows the user of the system to select an operation without worrying about messing up SRAM contents or anything during the setup and then starting the system once it's set up. Because of this, it is desirable to make the GO signal that the FSM sees be a single clock pulse wide; this process is referred to as 'edge-capture' because you are looking for a particular edge from the signal, not the actual signal itself. This isn't a particularly hard problem. Knowing that you want your signal to be a) synchronized, and b) a single clock pulse wide, try thinking of a design along these lines:

Start by synchronizing your signal, then add a device that will delay the
"signal by a clock cycle. No use the input and the output of that device," 
"combined with a little logic to create a single clock pulse wide signal." 

"The Walk-Request is a little harder. We need to latch this signal --- think"
"of it like a real walk button --- you want to push the button once, and then"
"have the traffic lights give you a walk session when the appropriate time"
"next occurs. Now, since you also want this to be synchronized, instead of"
"using a straight latch, think about using a flip-flop. Now, the question"
"becomes, how can you control a flip-flop to latch when it sees the WR signal," 
"yet can be reset by an FSM controlled signal?"

(\nodeLab2timing

"There are two timing issues in Lab 2, the first is how to construct a 1Hz clock" 
"and the second is what to make the system run at."

"The answer to the second question is easier, and much less constrained:" 
"basically, you have a bunch of digital circuitry that is capable of running at"
"very high frequencies; however, you are interacting with a world that is" 
"running at real-time. As such, your system clock should run fast enough" 
"that a person will not easily encounter problems, but slow enough to keep" 
"your circuitry functioning. This is a huge range. Most of the circuitry could" 
"handle the straight 1.8432 MHz (though the clock would need buffering); on the" 
"other hand, a person’s reflexes are only rated to about a twentieth of a second" 
"so anything over 25 Hz should be fine (25 Hz is pushing it, but certainly" 
"anything over 50 Hz is easily alright). I’d recommend something between" 
"100Hz and 10kHz."

"Do, however, keep in mind that a good selection of your system clock will make" 
"some other parts of this lab, namely the counter and the 1Hz clock, much" 
"easier to design."

"As for the 1Hz clock, this should be derived from your system clock. Give some" 
"thought as to what it should look like: should your clock be a 50% duty cycle?"
"Should it be a single pulse every second? Answers to these questions will help" 
"you to both design this clock and the counter."

(\nodeLab2counter

"The key thing to remember about the count(down) timer is that you are probably" 
"using a synchronous device. Typical chips to use include the LS163, and the" 
"LS169, both of which are synchronous, meaning that they need to see a _rising_" 
"clock edge to load. Now, when you assert your Starttimer signal, you should" 
"load a value into the timer, and, as was just pointed out, this requires that" 
"a rising edge be seen on the clock pin of the device."

(\nodeLab2fsm

"Designing the FSM of Lab 2 should be fairly straight forward. A partially" 
"complete FSMC file is given to you which should point you in the right" 
"direction."

"Otherwise, think about what needs to happen: you are going to start up your" 
"kit and hit reset to bring yourself to a known start location. From there," 
"you need to be able to handle whatever function is specified by the function" 
"switches when GO is received. In other words, you should start, as always,"
"at the beginning --- make a start start which is reachable from anywhere" 
"by pressing a RESET button. From there, you should watch for the GO signal."

115
"and, upon seeing it, determine the function that was called and then process"
"that request. Figuring out the functions themselves is up to you."

(make-node 'nodeLab3
nil
nil
)

(make-node 'nodeLab3timing
nil
"("Even as Lab 2 increased in design requirements over Lab 1, so does Lab 3 scale"
"upon Lab 2. Which of the following subblocks of the Digital Filter is"
"giving you difficulties:"
"
" [T]iming Block (including SYSCLK, UARTCLK, and SAMPLCLK)
" [A]2d Block
" [U]art
" [B]lock Decoder
" [S]torage Block
" [A][r]ithmetic Block
" [D]2A Block
" [O]ther
"

)(t {nodeLab3timing})
(a {nodeLab3a2d})
(u {nodeLab3uart})
(b {nodeLab3block})
(s {nodeLab3storage})
(r {nodeLab3arith})
(d {nodeLab3d2a})
(o {nodedesignother})
)

(make-node 'nodeLab3a2d
"("The design of the A2D in Lab 3 is not complicated, but it does need to take"
"into account a few needs of the system. Namely, the entire block can be"
"comprised of the AD670 chip; however, care needs to be taken as to wire"
"the chip consistently. In particular, note that you will need to be able to"
"control whether the AD670 is driving the data bus (you do not want bus"
"contention), and that you wire it such that it outputs data in a format"
"that matches the rest of your system. Also be sure to understand how to"
"initiate and control the conversion process."
""
"
)(t {nodeLab3a2d})
(a {nodeLab3a2d})
(u {nodeLab3a2d})
(b {nodeLab3a2d})
(s {nodeLab3a2d})
(r {nodeLab3a2d})
(d {nodeLab3a2d})
(o {nodeLab3a2d})
)
The UART is, perhaps, one of the most complicated chips you will have a chance to use in 6.111. For Lab 3, you do not need to design much with the UART as the lab gives explicit wiring instructions. However, the chip is complicated and a number of problems are associated with miswiring it. Are you having a problem that you are sure is connected to the UART? [y/n]

The block decoder module of Lab 3 is also fairly simple to make. Basically, the block decoder needs to consist of sufficient logic to produce two signals for your microprogrammer—namely a signal indicating the start of a block and a signal indicating that a response filter index matches the number on the switches. The key to designing this block successfully is to recognize the limitations of PALs. PALs can implement all the logic you need to do this; however, bear in mind that the signal indicating a match is likely to have a large number of terms in its MSP. You therefore need to find a PAL that can handle that many terms on an output pin.

The storage block is largely defined within the Lab 3 handout, so you should be sure to read the sections in that document before proceeding. A key idea for this block is that not all of the signals listed in the Lab 3 handout are needed for successful operation of the storage unit. True, all of the could be used in one way or another, but not all of them are necessary for successful operation; as such you should put some effort into considering which signals are truly necessary and which can be eliminated. Also bear in mind that the Storage block does drive the data bus and thus care needs to be taken that it does not do so all the time, else bus contention will develop.

Like the Storage block, the arithmetic module is partially defined in the Lab 3 handout. Again, reads the sections of the document that pertain to the arithmetic unit before beginning to elaborate on it—it really helps to know what you need to do and what has already been set up for you. The next major step is to go and read the man pages for 'mtable' and 'mulprom'. These two programs generate data files for your multiplication EPROM, and as such, successful construction of the arithmetic unit depends upon you understanding how these programs work and how you should incorporate them into your module. Also, you should not be afraid to ask for more chips from the desk. It is often during this part of the lab that students start to realize that they're running low on some kinds of chips—that's okay! If you need more '163s or '283s
"to implement your module, just go ask the desk for some more."
)
nil
nil
)

(make-node 'nodeLab3d2a
'("The D2A block, like the A2D, can be implemented solely with an AD558 chip."
"Care should be taken to understand how the AD558 works (so read that spec"
"sheet), and also you should be careful that the output of your arithmetic unit"
"meshes with the input format of the D2A unit. This often requires a little bit"
"of manipulation to get it right."
)
nil
nil
)

(make-node 'nodedesignother
nil
'("A number of other problems and topics come up during the course of 6.111"
"and the exploration of digital design in general, and not all of these rest"
"easily into a single, or any, of the labs performed in 6.111. As such, a"
"list of related design issues and problems follows, pick the one which seems"
"to be bothering you:"
"[B]us Contention"
"[N]ot listed"
)
'(b {nodebaddatabus})
(n {nodedesignheuristic})
)

(make-node 'nodebaddatabus
'("A bad data bus, or any bus for that matter, is often caused by bus contention."
"This is where more than one device that is connected to the bus, and capable"
"of driving the bus, is attempting to drive data onto the bus at the same time."
"This has certain inherent problems. Namely, a best, one of them will win, and"
"if that is the case it will almost undoubtedly be the one you don't want to"
"win. Much more likely, though, is that each will win some bits, and the"
"data on your bus will appear to be complete garbage."
""
"This problem is solved by the use of output enable signals, or line drivers."n
"Most of the complicated chips (A2D, SRAM, etc) will have some way of leaving"
"their output pins in a 'high Z' state, or a metastable state that leaves them"
"undriven, usually involving an output enable signal or sometimes a chip enable."
"This allows other devices to drive the bus without interference. For those"
"chips that do not have such pins, a line driver is necessary; usually an LS244"
"or an LS245 is used for this purpose, the result is the same --- the output of"
"the chip or switches is wired to the input of the line driver, and the output"
"of the driver can either let the data through, or leave the output pins in a"
"high Z state."
""
"Care should be taken to avoid bus contention. It is an insidious error, but"
"one fairly easy to avoid if you are aware of it."
)
nil
nil
)

(make-node 'nodedesignheuristic
'("At this point, your problem is a bit too complicated to try and match all the"
"details and categorize them. Get a TA a ask them to help you figure out what's"
"going on in your system and how you might design a circuit that does what you"
"want it to."
)
nil
nil
)
Appendix B

Supervisory Program Code

Listings

The following source code forms the knowledge base of the RTFM system. This code may be retrieved via anonymous ftp from sunpal2.mit.edu:/pub/digital_tools/. These files, as well as the files for the RTFM knowledge base, are in the RTFM.1.tar archive file.
### B.1 Frfun.l

This program is designed to work with the FROB system to create the frame object and several abstracted functions for the 6.111 Lab Expert project. This is a part of the M.Eng thesis by Owen Wessling, M.Eng EECS '96, MIT. Copyright (c) 1995 Owen Wessling All Rights Reserved.

This software is supplied "as is," without any kind of warranty whatsoever, but not limited to the implied warranties of merchantability and fitness for a particular purpose. The author and MIT make no representations about the suitability of this software for any purposes and do not promise to support it in any way.

This software is not to be incorporated into a commercial product, or access to it sold, without written permission from MIT. Otherwise, copies which include this notice may be freely made and distributed.

---

First—first we should create an environment object that holds a list of questions and a list of links. A loop of some sort should act as the main program, taking the first elements off of the question and links lists and asking the questions, and then comparing the user's response to the links and then activating the appropriate nodes. Those nodes' daemons will add more questions and links to the environment queues, and the loop will continue in this fashion. Should the lists ever become empty, then a simple message telling the user to refer to a TA might be in order.

```
(def-class memory nil
  ;;; make memory object; no inheritance;
  :slots ((queries nil) ;;; give it three slots:
    (links nil)
    (activated nil)))
```

We also want to define a basic memory object. This can always be redefined, but better to define something here, too.

```
(new-instance {class memory}
  ;;; make an instance called 'mem'
  :name mem)
```

Then, we create the node object. This is the heart of the frame knowledge base, as it is within these nodes that information and questions reside. Each node requires the following slots:

- **id**
- **state**
- **info**
- **query**
- **links**

```
((y {node18} {node17}) (n {node24} {node18}))
```
(def-class node {class daemon} ;; make the node class; inherits: daemon
:slots (id ;; 5 slots:
    (state 'inactive)
    (info nil)
    (query nil)
    (links nil)))

;; We will assume the memory object is called "{mem}" 
;; Also note that the current queries and links lists in {mem} are STACKS

;; DAEMONS:
;; Daemons should be defined over the state slot, set to print
;; the info slot, then add the query question and links to the
;; environment's query and link lists. Also, upon activation,
;; the name of the newly activated node should be appended to
;; the activation slot of mem.

(define-daemon ({class node} act-daemon state :after-assert)
  (if (eq $value 'active) ;; activates after-assert, checks if
      (progn ;; state is now 'active
        (if (not (equal nil (info $self))) ;; if info
            (if (not (equal nil (query $self))) ;; if query
                (assert-val {mem} 'queries (cons (query $self) (queries {mem}))) ;;; stacks
                (assert-val {mem} 'links (cons (links $self) (links {mem}))))))
        (assert-val $self 'state 'inactive) ;; then node can be returned 'inactive
          (assert-val {mem} 'activated (cons (id $self) (activated {mem}))))))

;; We can note here that RTFM has, in fact, two means of resetting the
;; knowledge base. First is that it never really needs resetting ——
;; since the daemon resets each node (making it inactive) whenever
;; the node becomes active. Second, we keep the activated list
;; around and can call 'resetRTFM' to traverse that list and make
;; all those nodes inactive.

;; In truth, both do not need to exist. Or rather, both do not need
;; to reset the nodes —— 'resetRTFM' also resets the memory stacks
;; and that will always need to be done. Still, 'resetRTFM' was put in
;; place before the multiple answers ability; and it is that ability that
;; dictates that a node reset itself immediately. (Consider the
;; possibility that a node is activated from two different directions,
;; we'd like the student to deal with the node both times and the
;; daemon will only initiate action if the node _changes_ to active;
;; thus, we reset each node after it does its thing.)

;; Now, we should also provide for a simple make method:
;; (make-node _id _info _query _links) should create a node of name
;; "_id" with id _id, list of info _info, and query _query.

(defun make-node (id info query links)
  (let ((inst (new-instance* {class node} :name id)))
    (assert-val inst 'id id) ;; Make an instance
    (assert-val inst 'info info) ;; then stuff the
    (assert-val inst 'query query) ;; the slots
    inst))
(assert-val inst 'links links)))

;;; A simple abstraction barrier: the prt function is defined to
;;; allow printing of a string. At first this in nothing more than
;;; a redefining of format, however in the future it could expand into
;;; Tcl/Tk supported window printing...

;;; (prt string) should result in the printing of string _string_

(defun prt (string)
  (eval (format t string)))

;;; sir

(defun query-user ()
  (read))

;;; simple text output

;;; simple text input

(defun resetRTFM ()
  (prt ""”Refreshing knowledge base and memory...””)
  (assert-val {mem} ‘activated nil)
  (assert-val {mem} ‘queries nil)
  (assert-val {mem} ‘links nil)
  (prt "...Done"”))

;;; tell the world what your doing

;;; then clear all the memory stacks

;;; just in case

;;; say you’re done

;;; Revexplode is the key to being able to enter multiple answers. It
;;; takes a single symbol response (like that from ‘read’) and converts
;;; it into a list of strings containing only the answers between ‘8’
;;; separators. It also reverses the order of the input (thus, a simple
;;; mapcar car take care of the activations in the right order).
;;; This list is returned.

;;; The first thing revexplode does is check to see if the input appears
;;; to be an integer. This is because the ‘string’ function has
;;; difficulties trying to convert numbers, so we have to shunt those
;;; off and turn them into a normal string manually and then come back.
;;; Then we can proceed as normal.

;;; Revexplode works by running a short loop in which it advances two
;;; counters. The first, i, keeps track of the beginning character
;;; of this entry; the second, i, is placed at the next occurrence
;;; of the ‘8’ character. Then we take the subseq of letters between
;;; the two counters and cons that with a running tally. When the
;;; loop collapses, because of no final ‘8’, we take the final subseq
;;; beginning at i to the end of the sequence, and we’re done.
;;; When making the tally, we cons the new with the old, therefore
;;; reversing the order of appearance.

(defun revexplode (resp)
  (let ((i 0) (l 0) (temp nil))
    (if (integerp resp)
        (setq resp (format nil "-d" resp))
        (setq resp (string resp)))
    (loop
      (if (eq (setq i (position \\& resp :start l)) nil)
        ;; define a few temp vars
        ;; make sure resp is a string
        ;; start loop
        ;; if no more ‘8’s in resp then

123
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Commandp tests to see whether response is a system command. If it is,
;;; commandp performs the function and returns a 1, otherwise, commandp
;;; does nothing else and returns a 0.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; We check for each possible system command in order, in a successively
;;; nested if clause:
;;; First we check for the help option (also called with "?") in which
;;; case we want to call the help function.
;;; Second, we test for the quit option; in which case we need simply
;;; clear the memory of all queries and links. Then the next loop will
;;; act as if the RTFM system had exhausted the knowledge base.
;;; Third, look for the backup command, which if called indicates that
;;; the user would like to terminate this branch of the search and
;;; backup to the last offshoot.
;;; Fourth, we check to see if the input looks like an activate command.
;;; If the command does not look like itself with the "$ACTIVATE_" string
;;; removed from the left side, then clearly it _is_ an activate command,
;;; so we want to change the state of the indicated node (assuming it exists)
;;; to be active.
;;; In any case, if one of the above is found, return that that statement
;;; was a system command, otherwise return 0.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun commandp (resp)
  (if (or (equal resp "?") (equal resp "!HELP"))
    (progn
      (helpcommand)
      1)
    (if (equal resp "!QUIT")
      (progn
        (quitcommand)
        1)
    (if (equal resp "!BACKUP")
      (progn
        (backupcommand)
        1)
    (if (not (equal (string-left-trim "$ACTIVATE_" resp)
                    resp))
      (progn
        (activatecommand resp)
        1)
    nil))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Helpcommand is simply the function that prints out the help
;;; information for the RTFM system.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun helpcommand ()
  (print "$Welcome to the RTFM system. This program will attempt to help you$")
)
In addition, RTFM supports a few special system commands:

- You may end your session at anytime by typing '!quit' at any prompt. This will end your current session, though you will be asked if you wish to quit RTFM completely or simply reset it.

- You may also activate a node directly with the command '!activate'. This command requires the identification of the desired node, which is attached to the activate command by an ' _ '. Thus, if you wished to activate a node called 'foobar', you would use the command '!activate_foobar'.

- Most nodes' names are not readily available; however, they are not hard to guess. Chips, in particular are straightforward, prefixing the chip name (LS04, 22v10, etc."

- Information on the LS151 is 'nodeLS151' and the activate command can be issued by '!activate_nodeLS151'.

- Also, you can access this help screen using either the '!help' or '?' command.

---

(defun quitcommand ()
(assert-val {mem} 'queries nil) ;;;; empty the queries and links
(assert-val {mem} 'links nil) ;;;; memory stacks and RTFM ends the session
)

(defun activatecommand (response)
(if (get-frob (string-left-trim '!ACTIVATE_' response))
(assert-val (get-frob (string-left-trim '!ACTIVATE_' response)) 'state 'active)
)

(defun backupcommand ()
)
The means to backup a level in RTFM is to simply drop the current
wusion and links on the memory stacks, so we just axe the
lists by their first element.

(defun backupcommand ()
  (assert-val {mem} 'queries (cdr (queries {mem})))
  (assert-val {mem} 'links (cdr (links {mem}))))

(defun expert ()
  (loop
    (resetRTFM)
    (assert-val (nodeRTFMstartup) 'state 'active)
    ;; this starts the first loop
    ;; then clear the stacks
    ;; activate startup
    ;; this goes to last branch

B.2 Super.l

; File: super.l
; Description: Supervisory program for 6.111 lab expert.
; Author: Owen Wessling
; Last Mod: 12/6/95 v1.0

This program is designed to work with the FROB system to create
the frame object and several abstracted functions for the
6.111 Lab Expert project. This is a part of the M.Eng thesis
by Owen Wessling, M.Eng EECS '96, MIT.
Copyright (c) 1995 Owen Wessling All Rights Reserved.

This software is supplied "as is," without any kind of
warranty whatsoever, but not limited to the implied
warranties of merchantability and fitness for a particular
purpose. The author and MIT make no representations about the
suitability of this software for any purposes and do not promise
support it in any way.

This software is not to be incorporated into a commercial
product, or access to it sold, without written permission
from MIT. Otherwise, copies which include this notice may
be freely made and distributed.

This program simply defines the initial function which starts the
expert system.

Basically, the function activates the initial node, which should drop
a question into the memory's query list and a set of links into the
links slot. From there, we simply do a while loop, until the
queries list is empty where we ask the question, then compare
the response to the link answers. When queries is eventually nil,
suggest a reference to a TA.
(loop
  (if (eq (queries {mem}) nil) (return)) ;; now start the query-answer loop
  ;;; break loop if no queries
  ;;; if the queries list is empty, we suggest finding a TA, because the expert
  ;;; has eliminated all threads in its knowledge base. Otherwise, we begin
  ;;; by first peeling off the first question and its links:
  (let ((question (car (queries {mem})))
        (answers (car (links {mem})))) ;; pick off the question
    ;;; and its links
    (let ((question (car (queries {mem})))
          (answers (car (links {mem}))))
      ;;; now ask the question. remember that the question is a list of strings, so
      ;;; we can print the whole thing by using mapcar on the list.
      (mapcar ' (lambda (x) (prt x) (prt "-") question) ;;; print the question
        ;;; now, if the links != nil read the answer from the user and process it.
        (if (not (eq answers nil)) ;;; if links exist, then get response
          (progn
            (prt "==>") ;;; give prompt
            (setq response (query-user)) ;;; get response
            (prt "-7©©©©©©©©©©©©©©©©©©©©©©©©©©©©©©©©©©©©©©©©©©©©©©©coo.") ;;; response is now a symbolic form from the read, but it may contain
            ;;; several responses. We want to call revexplode on it to turn it into
            ;;; a list of response strings.
            (setq response (revexplode response)) ;;; process response
          ))) ;;; now, answers is a list of possible solutions, we want to check
          ;;; each response to each possible response and then activate the proper
          ;;; nodes.
          (setq endanswer nil) ;;; temp variable
          (mapcar ' (lambda (x) (setq tempanswer answers)
                       ;;; for each response
                       (setq tempsanswer answers)
                       ;;; set temp to answers (links)
                       (do ()
                         ;;; if answers is nil or match, stop
                         (if (eq tempsanswer nil)
                            (if (integerp (caar tempsanswer))
                              (equal x (format nil "-d" (caar tempsanswer)))
                              (equal x (string (caar tempsanswer))))
                           (setq tempsanswer (cdr tempsanswer)) ;;; otherwise look at the next link
                           (if (not (eq tempsanswer nil)) ;;; if links not nil, then match
                             (setq tempsanswer (append tempsanswer (list (caar tempsanswer)))) ;;; add it to endlist
                           response)) ;;; map all responses
                         (setq answers endanswer)) ;;; then set answers = endanswer
          ))) ;;; after loop is finished, if answers is not nil, then the user's response
          ;;; was valid and we should continue, otherwise, drop it.
          ;;; Now, the answers list should contain a list of all of the accepted
          ;;; responses, each of which is a list of the form:
          ;;; (<response> {node1} ... {noden})
Thus, we need to map to each element in answers a function which will take the above format and activate the right nodes, which consists of mapping another function to perform

(assert-val (eval <element>) 'state 'active)

to each element in the cdr of each list (the nodes).

Of course, if answers is nil, then the response wasn't valid so we should check to see if its a system command, and in any case we should do nothing to the current memory stacks and simply try again.

If the response wasn't a system command, we should tell the user that they entered an invalid response.

```
(if (eq answers nil)
    (if (commandp (car response))
        (prt "Your answer was not recognized as an answer to the given question."")
        (assert-val {mem} 'queries (cdr (queries {mem})))
        (assert-val {mem} 'links (cdr (links {mem})))
        (mapcar #t (lambda (x)
            (assert-val (eval x) 'state 'active))
            (cdr x)))
    (progn
        (assert-val {mem} 'queries (cdr (queries {mem})))
        (assert-val {mem} 'links (cdr (links {mem})))
        (mapcar #t (lambda (x)
            (assert-val (eval x) 'state 'active))
            (cdr x)))
)
```

Now, that finishes the processing if answers wasn't nil to begin with.

If answers was nil at the very beginning, then we had a question slot acting as an information giver without a real question, so we just want to continue. Thus, if answers was nil at the beginning we just take the cdr of the queries and links list and continue.

```
(progn
    (assert-val {mem} 'queries (cdr (queries {mem})))
    (assert-val {mem} 'links (cdr (links {mem})))
)
```

Now that the loop is over, we should send a brief message saying that the question overextends the knowledge base and a TA should be consulted to help the student solve their problem.

```
(prt "%-%-%RTFM has depleted what it considers to be relevant questions."")
(prt "%Please take the time to think about your problem and try to figure")
(prt "%out why your problem exists. If you are still stuck ask a TA for help.")
```

```
(setq response (query-user))
(if (eq response 'y) (return))
```

B.3 RTFMinit.l
(setq RTFMdir "/mit/owenwess/thesis")

;;;; First load the PROBS platform (from the frobs/load.l file):

(load (concatenate 'string RTFMdir "frobs/pack.1"))
(load (concatenate 'string RTFMdir "frobs/globals.1"))
(load (concatenate 'string RTFMdir "frobs/struct.1"))
(load (concatenate 'string RTFMdir "frobs/misc.1"))
(load (concatenate 'string RTFMdir "frobs/define.1"))
(load (concatenate 'string RTFMdir "frobs/user-methods.1"))
(load (concatenate 'string RTFMdir "frobs/ask-assert.1"))
(load (concatenate 'string RTFMdir "frobs/daemons.1"))
(use-package 'frobs)
(provide "frobs")

;;;; Now, load the RTFM files:

(load (concatenate 'string RTFMdir "sources/frfun.1"))
(load (concatenate 'string RTFMdir "sources/super.1"))
(load (concatenate 'string RTFMdir "sources/chips.1"))
(load (concatenate 'string RTFMdir "sources/design.1"))
(load (concatenate 'string RTFMdir "sources/kit.1"))
(load (concatenate 'string RTFMdir "sources/tools.1"))
(load (concatenate 'string RTFMdir "sources/system.1"))

;;;; We now define the preliminary startup node for the RTFM system:

(make-node 'nodeRTFMstartup

"Welcome to the RTFM v1.0 system. This program is intended to help you"
"to debug and design your digital lab projects, particularly those"
"for 6.111.")
Any questions, comments, bug reports, or suggestions should be sent to owenwess@mit.edu.

Thank you for using the RTFM program, and I hope you find it helpful.

Copyright (c) Owen B Wessling, 1995. All rights reserved.

The RTFM system works by asking you to identify what sorts of problems you are experiencing and then directs its knowledge activation according to your answers. As such, answer any questions to the best of your ability at the time. Feel free to send any comments about confusing sections to the above individual.

Also, keep in mind that this is a tool for educational purposes. It is not designed to give you answers to your problems, but rather to give you sufficient information that you can either deduce the solution or find the problem and correct it.

Now, which of the following areas do you need help in:

- You need information about a specific chip.
- You are experiencing problems with the Nubus labkit.
- You need design help.
- You have a problem using a particular (software) tool.
- You need general help debugging your system.

Now, start the system

(expert)

When done, leave.

(bye)
References


