DEVELOPING
A COMPREHENSIVE SOFTWARE ENVIRONMENT
FOR PASSIVE SOLAR DESIGN

by

STEVEN E. LOTZ

B.A. in Architecture
University of Washington
Seattle, Washington
1982

SUBMITTED TO THE DEPARTMENT OF ARCHITECTURE
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE
MASTER OF SCIENCE IN ARCHITECTURE STUDIES AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUNE, 1985

Signature of the author [illegible]
Steven E. Lotz
Department of Architecture
May 9, 1985

Certified by [illegible]
Harvey J. Bryan
Assistant Professor of Building Technology
Thesis Supervisor

Accepted by [illegible]
Julian Beinart
Chairman
Departmental Committee for Graduate Students

DEVELOPING A COMPREHENSIVE SOFTWARE ENVIRONMENT FOR PASSIVE SOLAR DESIGN

by
Steven E. Lotz

Submitted to the Department of Architecture on May 9, 1985
in partial fulfillmant of the requirements for the
Degree of Master of Science in Architecture Studies.

ABSTRACT

This thesis is a journal which describes the thoughts and decisions leading up to the final design of a comprehensive software environment for passive solar design. The main purpose of this writing is to convey why a comprehensive software environment for this particular field is needed in order to help teach the principles of passive solar design, so that they can be adequately taken into consideration in the architectural design process, and how such a system could be implemented.

A case study involving the use of previously available passive solar design tools is used to point out areas where these tools are deficient in their ability to focus a designer's attention on pertinent building performance simulation data, which could be more effectively used to influence design decisions at the various stages of the design process. This leads to a discussion of how these shortcomings could be overcome through a new and different software design strategy which utilizes a systems approach to build a more flexible and powerful passive solar design tool. Through further experiments, practical considerations and real-world constraints are brought to light, and how they affected the conceptual development of such a system which I undertook to develop here at MIT for Project Athena.

Next, certain implementation details are given which seek to bridge the gap between conceptual goals and practical software design considerations. How the internal organization of software code affects the external interactions between the user and the system, and how it can promote the qualities needed for software survival in an educational setting is addressed. Finally, the outcome of an experimental prototype for this system is discussed, as well as my concluding thoughts regarding what I have learned through this endeavor about writing architectural design tool software.

Thesis Supervisor: Harvey J. Bryan
Title: Assistant Professor of Building Technology

ACKNOWLEDGEMENTS

Harvey Bryan	For his committment toward providing an open educational environment for his students which encourages individual experimentation, and for his support as an academic and thesis advisor.
Project Athena Staff	Namely Joseph Ferierra, who saw the possibilities this project possessed, and helped to push it through Athena.
Flagship Crew: Steve Ervin Peter Jorgenson	These particular guys were very generous with their time when I became too exasperated to debug code by myself.
Bill Wright	For the opportunity for some real-world C-programming experience, and his effort to provide valuable input on this thesis.
Ron Fergle	For his enthusiasm and determination to carry on this work via adding enhancements. This thesis was written for students like Ron.
Warren Belfer Robert Haley	Warren and Robert contributed the original code this work is based on. Many thanks for your generousity.
James Griffith David Cistola Rick Riemersma Scott Nason Joe Nadoe	This work wouldn't have suceeded with without the spiritual involvement of these members of the Park Street Church whose prayerful concern before God on my behalf gave me the vision, the insights, and the perseverence I needed in order to pursue this work. Keeping me centered on the spiritual goals of life during those bogged-down moments when this was hard to do. This year of my life is indebted to you all!

TABLE OF CONTENTS

INTRODUCTION

It seems only fair to me that I take a few moments to try to inform the reader of what this thesis is really about, before he or she commits themselves to reading it. I have picked up many a thesis myself, only to find out halfway through reading it, that although it did contain information on the general topic I was interested in, it wasn't quite what I was looking for, so I had to go off in search of another which was geared more toward my specific needs. The title of this thesis is "Developing a Comprehensive Software Environment for Passive Solar Design". While I will indeed be talking about the nature of a comprehensive software design package for this purpose, I must stress that the key word in this title, is the word "developing".

What I am going to attempt to do is to convey what I have learned over the last several years about developing architectural software, in particular, as it relates to the special sub-field of architectural design commonly known as "energy-conscious design". My experiences in this realm have been mostly experimental in nature having come from many trails and errors, as opposed to a more humane "book-learned" approach, although I do claim to have done some reading on the subject of software design. The fact is however, as most of us working in this area concede, the use of computers to aid design, while not new to the engineering profession, by and large is new to the architectural profession. While many in our profession now groan at this reality, now that we are all faced with the seemingly horrendous task of becoming computer literate in order to keep up in a competitive marketplace, it has afforded us the relevant excuses to go back to "ground zero", and seeing the somewhat negative experiences others in other professions have had regarding computers, to rethink how computers can best be utilized to advantage. I will have much more to say on this topic in the next chapter, I just want to lay out the scene for a moment so that the reader understands why this work is so experimental in nature, even though we all know that computers have been around for quite some time. I think it is fair to say that our profession over the years, most probably because we are so visually oriented, has not been that impressed with the quality of the computer user-interface as has been witnessed by the use of computers in other professions. So we have taken the opportunity to experiment upon ourselves in the hope of rectifying this situation somewhat, particularly in view of the fact that recent hardware and software advances at the commercial level offer us a virtual guarantee that our time spent doing this is liable to pay off.

At any rate, since my learning process has been so experimental in nature, if I am going to communicate what I have learned to the reader regarding my thoughts on software development in this field, by necessity I must go into the details of some of these experiments. Consequently, this thesis is not the typical sort of engineering treatise where one examines some phenomenon in light of someone else's theories in an effort to convey how carefully one has read the textbooks. Nor am I trying to defend my knowledge as if I had the final say on some pertinent questions. What seems most important to me, is that I convey to the reader in a very personal light, how I went about doing the research in software development that I have done in the past few years, in order to bring to the surface the beginnings of a personal design philosophy regarding software development in this field. Being able to purport a design philosophy is after all the true test in my mind as to whether or not a person has gained anything of value from an architectural education- which is what I have been endeavoring to do these last five years of my life. With this goal in mind, the reader will soon become aware of the very personal, conversational style of this writing- as opposed to a more standard, technical tone. In essence, this thesis is really a personal journal of my attempts to get a grip on what software development in the architectural field is all about. Although I hope this makes this thesis more palatable than some I have had to read recently in preparation for my own, the main reason I have chosen this particular writing style, is so that I can attempt to convey the behind-the-scenes reasons why my work took the particular direction that it did. Since no design philosophy can ever be "thee correct design philosophy" so to speak, since there are always so many available avenues to take- several of which may lead to the same place, I hope this inside glimpse will help the reader understand how I have arrived at the particular conclusions that I am beginning to arrive at, through this experimental learning process.

To this end, the hypotheses that drove me to do this research, rather than being explicitly stated here, will be unveiled gradually throughout the earlier chapters as I get to the particular points in this odyssey where they became issues which subsequently affected my journey. In this way, the direction of my thinking should be clear to the reader, and the reader will be better equipped to understand the particular course of action that I eventually was to take as a means of finding a solution to the software development problems I foresaw. As I will state in my conclusion, I think this particular style of thesis writing is not only beneficial in terms of my goal- to communicate well to the reader, but also a necessity, and a requirement of my final hypothesis, that the process of software design is not a straightforward linear process as the "engineering mind" would like to believe, but in fact is a "wicked design problem" very similar to the architectural design process (1). As such, I do not believe there is one right answer to the problems I have attempted to solve in this thesis. The particular direction I have taken is only one of several that I could have taken, and I took the one I did because I felt I was best equipped to take it- meaning that I felt it would be the most successful direction for me to go. As the reader will soon find out, my superiors on this project were all the while purporting a different solution strategy, which I did not buy as the one which looked the most promising- which is not to say that I think their's is in any way "wrong". Therefore, because I believe the software design process, on the scale that I have undertaken in this project is a "wicked design problem", (scale is important- i.e.- one would hardly consider a garage to be a "wicked" architectural problem, but most large buildings certainly are), this thesis is not trying to "prove" anything in the traditional sense, except that this needs to be taken into account in one's design approach. In fact, this thesis may very well leave more questions unanswered than it does answered. Such is the very nature of this type of problem, which is why they are so challenging to those who have the "architectural mindset", who have the breadth needed to realize that there are problems such as this, and therefore don't mind what others would consider to be inconclusive results.

Of course many of the factors that affected the subsequent direction of this work involved the real-world circum-stances of having to work with others at MIT, and in particular, those on the School of Architecture's Project Athena Staff, of which this project is a part. The reader will soon find out that I have attempted to be very open and frank about this part of my experience. Being in it's initial start-up mode, Project Athena was in a constant state of flux during the time of this software development effort. The many "changes of mind" and constraints which I had to undergo, greatly affected the avenue I was to eventually take. As such, not to recount these experiences would be detrimental to my current task. To anyone who may be in some manner insulted by my "up-frontness" in this regard, I offer my sincere appologies beforehand. Some people were forced to wear "Athena hats" so to speak, by virtue of their role in this project, and any antagonism felt by anybody through my words in the following chapters must necessarily take this fact into account. My arrows were all aimed at the hat, and I am truly sorry if anyone feels that I missed my aim. It's not easy to take these kinds of necessary shots!

For now, I want to express my sincere opinion that if the reader finds any value out of what I will have to say in this thesis, it only goes to show that others working before me (and particularly here at MIT), have left me a rich legacy to build upon. Much of it- even though some of it was done years ago, still remains noteworthy today. Not too long ago, a fellow colleaque Tim Johnson offered his assistance to me concerning my perceived need at the time to develop some elaborate data structures for this project. In the context of his offer, Tim mentioned that he was disappointed to learn that "things haven't changed that much since the early work I was involved in with Sketchpad (2)- at least as far as data structures are concerned- and that was back in the late '60's!" By and large, Tim's statement is probably correct, which only goes to show that the early work of Negroponte, Purcell, and others was ahead of it's time. It has taken the rest of the world the intervening fifteen or twenty years to catch up, mostly in terms of having the equivalent of their computer power at a localized enough level to begin to take advantage of their earlier work.

A few years ago, namely as soon as cheap microcomputers began arriving on the scene, and others of us could finally get glimpses of these machines that we had long heard about, people here at MIT were again quick to jump at the opportunity to explore the use of the computer as a design tool for architectural purposes. Since these early microcomputers were definitely short in the area of graphics display capability, but at the same time were a marked improvement over what most had in the area of an affordable computational tool at the time- i.e.- programmable calculators, those who were not inhibited with the fact that the micro could not produce graphics began to experiment with this machine, trying to discover it's potential as a valuable design tool. As the reader might anticipate, the likelihood that these primative micros could be put to some good use rested with those within the engineering spectrum of the architectural field, who were not dealing with such elaborate computer algorithyms that they necessarily had to overlook the micro as a potentially valuable tool. Those who's needs rested primarily with a good graphics interface, or a highly efficient computational engine- such as those involved in structural design or computer graphics necessarily have overlooked the micro until very recently.

This was not so with those who were involved in the reinvented field of energy conservation and passive solar design which in the wake of the oil crises of the '70's, began to reemerge on a grassroots level during this period. By necessity, because of their somewhat precarious stature within the mainstream of the architectural profession, these people were in desperate need of a "poor man's computer", and as such in contrast to other factions within the profession, welcomed the microcomputer with open arms. I am not trying to give my sub-field within the profession at large the credit for being first to discover the power of the micro- I can just hear my colleagues in the urban planning department now!, and this is not my purpose for this somewhat lengthy discourse. (As an aside, I think they in a parallel fashion, probably discovered the micro for the same reasons we did.) My only point is to give a somewhat historical perspective to my field's introduction to the microcomputer, and to suggest that there is a rich albeit short legacy which this work in particular stands on.

I do not intend to duplicate the efforts of my predecessors- either in my research work or in this thesis but instead, refer the reader to the Chapter Notes, where the part of this legacy which pertains most directly to a more thorough understanding of this thesis can be found. Although I will gloss over the work of these people in the next few chapters in laying out for the reader the reasons why my own concerns took the direction they did, I refer the reader to these other references for much more detail. In particular, I encourage the reader to look closely at the previous work of Professor Harvey Bryan, and his compatriots, regarding the use of micros to do building energy analyses of which this work is a continuation thereof. The "et. all" in this case, refers mostly to the previous work of David Krinkle, Charles St. Clair, Constantine Seremetis and Makoto Hoshino, whose explorations down various avenues- whether they bore immediate fruit or not, certainly planted the seed by which future research continues. I consider myself indebted to these people, as well as to my other colleagues on the School of Architecture's Project Athena staff, for what has turned out to be a rewarding educational journey here at MIT. Not that MIT deserves all the credit for this. As the reader will soon find out, my educational journey in this particular realm started somewhere else, and at times has taken some unexpected twists, most importantly, an opportunity to work for William A. Wright Associates last summer, on a rather large, "real-world" software project. That experience turned out to be invaluable in terms of helping me make decisions at specific turning points during the course of this most recent exploration. Not to mention the fact that Bill himself is a very enthusiastic and serious C programmer, the type that can't help but inspire a young upstart such as myself, and there was always the occasion to ask myself "gee, how would 'Mr. WAW' himself handle a situation such as this!"

But this is how we all learn- in the wake of others, which brings me to my last concern regarding this introductory chapter- which is why I have undertaken the task of actually writing this thesis. To be sure, the actual work behind this thesis is ninety percent completed by this time, and to say the least, I have never been of the type who could submit themselves to the requirements of formality- unless I could find an overriding better reason. MIT has recently inaugerated "Project Athena"- which reminds me of President Kennedy's national goal to put a man on the moon by the end of the sixties. In a similar way, MIT has an institutional goal to recomputerize itself (and therefore in a larger sense, revitalize itself) using state of the art computer network technology over a five-year period, And at this point, I imagine that the outcome of this present goal will be similar- i.e.- just barely scraping by,- only because it is a similarly ominous task.

The work behind this particular thesis represents part of the first wave of software projects aimed at making this goal a reality, and therefore- being such a small part of such a large project at such an early date,- in some ways it has necessarily been a "stab in the dark". Not much in the way of divine wisdom coming down from the goddess Athena above- to be sure! What the reader will find out later- is that due to the unfolding circumstances, (late delivery dates of hardware and the related software support), from day one- I didn't expect any either. Consequently this project for me has represented- on a larger scale than the opportunity afforded me last summer- the chance to take on a "real world" software project by myself, because the decisions that had to be made at every point were by necessity- business decisions as much as anything else, and I had to make them myself. Of course I wanted to anyway.

While the students involved in the next wave of projects may get alittle more guidance from "up above"- I suspect that to a large extent they will be forced to go it on their own for the same reason. Large facilities necessitate large beaurocracies- which can never keep up with the needs of an individual project, nor with the latest innovative idea in the mind of a creative software developer. And the logical corollary I insist on adding: If you wait until the beaurocracy can meet your needs, 1) within the time constraint that you have, you will get virtually nothing accomplished, and 2) your work will suffer the consequence of not being as creative and innovative as you would like it to be (which is a ruboff on how it has in turn affected you!) Witness the results of the first wave of Athena projects which I think bears this out. Consequently- like it or not, I know that many will have to follow in these footsteps, faced with the same choices and decisions along the way. Learning from experience in a trail and error fashion- which means learning how to recover gracefully and quickly from your blunders, just as I had to.

It is my hope therefore, that the largest contribution I will have made after I am long gone- is not the fact that I will have left behind alittle more software for the school's computer lab- but that this personal account of my own attempts to write software will have been an inspiration and a help to others. Indeed, if the software I have left behind is not outmoded and replaced within two years (especially considering it's experimental and therefore unsophisticated nature), I will be very dismayed and concerned about Athena's real effectiveness. With this perspective in mind, this thesis writing is dedicated to those who at this point, are contemplating following in these footsteps, in the hope that:

1) You won't be scared off.
2) Based on this forewarning, you will be able to avoid my mistakes at least, and will be better prepared to cope with your own.
3) You will have been more successful than I was able to be, in an equivalent time frame.
4) And finally- you will have learned as much as I did!

And I wish you the best of luck in your dealings with Athena!

CHAPTER 0: BEGINNINGS

"A battered reed He will not break off,
and a smoldering wick He will not put out
until He leads justice to victory."

This is chapter zero. Everything starts at zero, and I even dare to say that all C programmers who have ever translated a program into C from BASIC, Pascal, or FORTRAN know this. At least if they don't- I'll bet they're still working on the program! I say this for the benefit of those of you who are learning C and haven't yet had the opportunity to track down a pointer that just flew into "outer space" (the operating system). Usually disasterous consequences result when one forgets for even a moment that pointers need to be anchored to a beginning, and that arrays really start at zero, and not at one. Projects always have beginnings too. Back in '81 at the University of Washington, I happened to take a course in passive solar architecture with Joel Lakin, an instructor who ended up finishing his master's degree at UCLA by writing a program with Murray Milne called "Solar 5". By the time Joel arrived in Seattle, enough time had elapsed since his previous intimate aquaintance with the solar design tool world, that he- as well as those of us who- like him, had been previously introduced to the computer world via FORTRAN on large mainframe systems, had to be reeducated from zero!

The course was conducted as a studio, and on the first day of class, we were broken down into groups of three to work on a quarter long design project. The design problem was to come up with a passively heated house that could be built on any "standard" Seattle city lot with a southern exposure. The house had to be as energy efficient as possible, with the overlying constraint that it had to be able to be constructed (exclusive of the cost of the lot, but inclusive of some "sweat equity" on the part of the owner to do some unskilled work) for a total of $40,000. Each design team was respondsible for finishing their house- complete with working drawings, a cost analysis, and a final energy analysis by the end of the quarter when- providing things turned out well, and the instructor got his act together, the drawings would be bound in a "spec book" for public distribution. The object of this undertaking was to provide young couples in the Seattle area with an alternative toward the rising costs of housing- and utility bills, which were fast becoming unaffordable to those who were just starting out in the housing market.

What made this passive solar course unique (I had already taken two others), was the requirement that computer design tools be used during every phase of the design process- from the schematic design stage right on through the completion of the working drawings. Unfortunately because school resources were limited, we were not able to use the very latest design tools available. However, what we did manage to get proved to be perfectly adequate. During the schematic design process, we were able to use a "sun machine" designed by Professor Omer

Mithuen, as well as my HP-41c version of PASCALC, an SLR program developed by Total Environmental Action (1). During the design development stage we used TEANET, another TEA software product which was a forerunner of many of the subsequent microcomputer thermal network models, which ran on a TI-59 programmable calculator (2). For the final working drawing's energy analysis, we were able to "squeeze" the department enough to give us what was necessary to set up an account with Berkeley Solar Group whose mainframe program CALPAS3 at that time, was considered to be the "cadillac" in passive solar design tools (3).

At first, alittle confusion arose amongst the design teams, because although several students in the class had experience designing solar homes that actually got built, nobody had any experience using computer design tools in the process of design. Consequently the big questions that everybody had to deal with were- what tool do you use when ?, how does it get used to best augment the design process ?, and how much do you rely on it ? After some initial fumbling around, we all seemed to get the hang of it. At the onset, most groups used the sun machine and Balcomb's Rules of Thumb from the SLR method (4) to try to get a workable plan where the various design parameters were in the right proportion to each other- i.e.- the right percentage of glazing to floor area, which was consistent with the amount of mass and the particular passive system type chosen, etc .... Those of us who had access to PASCALC used it as well. These methods gave us an indication of whether or not a particular plan was amenable to our goal of providing a high solar savings fraction (5), along with the necessary architectural amenities to make it a comfortable and interesting place to live. After exploring various plans and discussing the many tradeoffs for a few weeks, we took a vote as to what seemed to be the most promising plan to develop. During the next phase which seemed to encompass most of the course, we used TEANET as a means of tightening up our plan, and making sure that no significant energy issues were being overlooked- such as overheating problems during the "swing seasons" when fixed shading devices could not cut out unwanted solar gain, etc ....

By this time, we had all discovered that design tools were indeed valuable tools, but that they couldn't be treated simply as "black boxes" that cranked out answers. That each design tool by way of the fact that it represented an abstract mathematical model, required a considerable amount of engineering wisdom when translating the various building components into program input quantities. Moreover, each tool was unique with regard to the algorithyms and assumptions that it rested on. Consequently, alot of adaptation was required in order to use these tools together. An example will serve to illustrate this point. PASCALC and CALPAS3 both had an ability to deal with overhangs, but TEANET did not. Due to the limitations in computing power of the TI-59 calculator, the authors left off this feature. The class operating budget did not afford us the luxury of using CALPAS3 (the ideal tool) during the design development stage, therefore TEANET had to somehow be adapted for the purpose. As it stood, TEANET was cranking out questionable results which did not seem consistent with our PASCALC results, because this feature was lacking. The model was seeing too much solar gain, and consequently telling us that our building was overheating during the warmer months.

We had reached the first big hurdle. What do we do- throw away the design tools, and go back to our old method of design ? This was of course a big temptation after finding out that design tools couldn't be treated as black boxes. Maybe we were just wasting our time. But a quick browse through the TEANET user's manual (you know- that document that may sometimes get used as a last resort!), told us that the program's limitations required that it have a "smart user", so we went back and tried again. Since PASCALC is a modularly designed program having a separate and accessible overhang calculation, we were able to apply it's overhang algorithym to TEANET's required input before processing.

In retrospect, considering the overwhelming success of the results of this project, I am very glad that we did this. Because the second temptation was to believe TEANET and not PASCALC, as TEANET- being a thermal network program gave us more discrete output information such as temperatures within the space. I think there is always the temptation to believe the program that on the surface due to the type of output it produces- looks more accurate. TEANET was programmed to spew out temperatures with two significant digits of accuracy- i.e.- "during the last hour, the inner surface of the trombe wall was 67.16 degrees fahrenheit". This seems more believable than "50% solar heated". If we had believed TEANET and not made subsequent compensations in it's input, our building would not have ended up as energy efficient as it did. So I'm glad that one of us did check the manual as a last resort!

Eventually we found out that similar inconsistencies existed between TEANET and CALPAS3. The latter had a provision for a ground temperature node, while the former did not. Since we weren't going to be doing any CALPAS3 runs until the end of the course, we weren't sure that this problem was that significant, but a simple check of giving TEANET a "fudged" ambient temperature value that was halfway between the real ambient and the ground temperature for a winter month, convinced us that we ought to do something about the discrepancy. Otherwise, which tool were we going to believe in the end ? Again alittle ingenuity proved adequate, and we stuffed an additional resistance between the outside surface node of the slab and the ambient temperature node. While this wasn't exactly kosher because of the lack of a ground capacitance, the value we used for this resistance was derived from Balcomb's BLC calculation routine for perimeter slab loss (6), and therefore we made the assumption which seemed logical to us, that the effect of ground capacitance had been taken into account. If not- at least it was then Balcomb's problem, and in the meantime it made our TEANET runs consistent with our PASCALC runs.

By now, I'm sure the reader is wondering whether we found all this hassle worth it. The answer is an overwhelming "yes". I already stated that several in the class had previous experience designing actual- not just "academic" solar houses. Consequently, many of us considered designing solar houses to be "basically simple- just follow Mazria's Rules of Thumb (7), and you've got it made!" Not so simple, we found out. Design tools proved two things that the seat-of-the-pants method of design had not already proved to us. And boy, were we glad to make these mistakes on paper rather than out in the real world like most architects! Design tool rule no. 1 was that if you incorporate anywhere near the amount of glazing that you would like to use without adequately compensating for it (a difficult task)- you are going to have "roast occupant" for a July 4th barbecue! Notice that I said "anywhere near the amount of glazing". Our group ran a total of 25 runs using TEANET, of which 10 were final runs of legitimate plan variations. The reason ? For all groups, the main design problem for the course turned out to be- how do we accomplish our stated goals without having our building end up looking like a standard tract-home- i.e.- that has walls with a few little "punchouts" for windows. In other words, the overheating problem constantly defied us the opportunity to design a passive solar house that had enough south glazing to look like one, and thus "spoke" solar.

As if this weren't bad enough, on the other end of the spectrum, design tool rule no. 2 was that if you didn't find a practical alternative to night insulation- all the heat that you gained during the day, you were going to lose at night. Again, notice that I said "practical alternative"- because in many cases, night insulation is either too impractical or too costly, but in most cases- both. You can go without it, but since our design program required that our building be as efficient as possible- we didn't dare risk the chance that one of the other groups would find a decent alternative.

All of us were overjoyed with the outcome of the results- that with all the fiddling around that took place to make things work, that (if I remember correctly) for the most critical case, the results from CALPAS3 for peak temperatures were within a degree fahranheit of those predicted by the equivalent TEANET run. Not to mention the fact that CALPAS3 predicted that the building would require only $64 worth of backup electric heating at the greater Seattle electric rate without consequential overheating of the interior of the building. But we were using wood for backup heat anyway. (The sunspace would overheat, which was of course intentional on our part, as a means of dumping alot of excess heat when we didn't want it. Since the sunspace could be closed off from the rest of the house, we considered this o.k. And by the way I forgot to mention- since TEANET was a one-zone program, another necessary adaptation during this experiment was to try to model this building- sunspace included, as a direct gain space. Of course, this required some more engineering guess-timation, so we just figured that at a certain point the building would just start dumping excess heat, but otherwise this slight-of-hand procedure wouldn't affect things too much.)

At the end of this class experiment, the class as a whole got together to evaluate their experience. Everybody concluded that their knowledge of passive solar design had be raised considerably through the use of these design tools. Both from our own experience in having to put them to use, and in seeing how they had affected the outcome of other projects as well as our own. For example, we all learned- without having to make the mistake of trying one, that trombe walls don't work well in Seattle. There's just not enough sun to get that mass heated up, so you are much better off getting what little solar gain you do have into your living space via a direct-gain or sunspace system. We all concluded this because none of the houses with trombe walls achieved as good results from either network simulation program.

Much discussion has arisen amongst those in the design professions as to the role design tools ought to play in design practice. Some say that design tools should be educational tools only- as vs. design tools which are used in everyday practice. The promoters of this theory believe that the designer after using design tools for awhile will begin to "internalize" what knowledge he has gained from the tool, and therefore after alittle experience with it will no longer need to use it, except for that once in a blue moon oddball situation (8). In light of the above outcome, I find this argument to be very moot. One will never know what he doesn't know until something shows him that he wasn't as smart as he thought he was. In this particular case study, many of these students had previous experience designing passive solar houses, and were at the onset of our experiment, alittle skeptical as to whether or not their time was going to be well spent. In 20/20 hindsight, all felt that they had learned alot through this experience. Of course I must remember that we were students. Perhaps a designer with alot of experience would find little use for a design tool. I suspect the real answer would lie with the amount of risk an architect is willing to endure while putting his reputation on the line. Would one of these proponents be willing to design a building such as the one used in this case study (say for argument's sake that the client was the City of Seattle)- with similar design objectives and guarantee that it would work, without double checking his "designer's intuition" with a design tool? (If he says "yes", he's either got lots of malpractice insurance or he's going to call his engineer- who will use a design tool!)

The real point to be grasped I think, is that whether or not a design tool is used in the classroom or on the job, it should be designed as a good learning tool- as verses a final check, production oriented tool, because that is it's main value in either case. The process of designing wherever it takes place, is primarily a learning process by virtue of the constant feedback required to keep it going. For a good discussion of this issue and it's relevance to design tool design, I refer the reader to the reference in the Chapter Notes by David Krinkle (9). For my purposes, a more pragmatic discussion is in order.

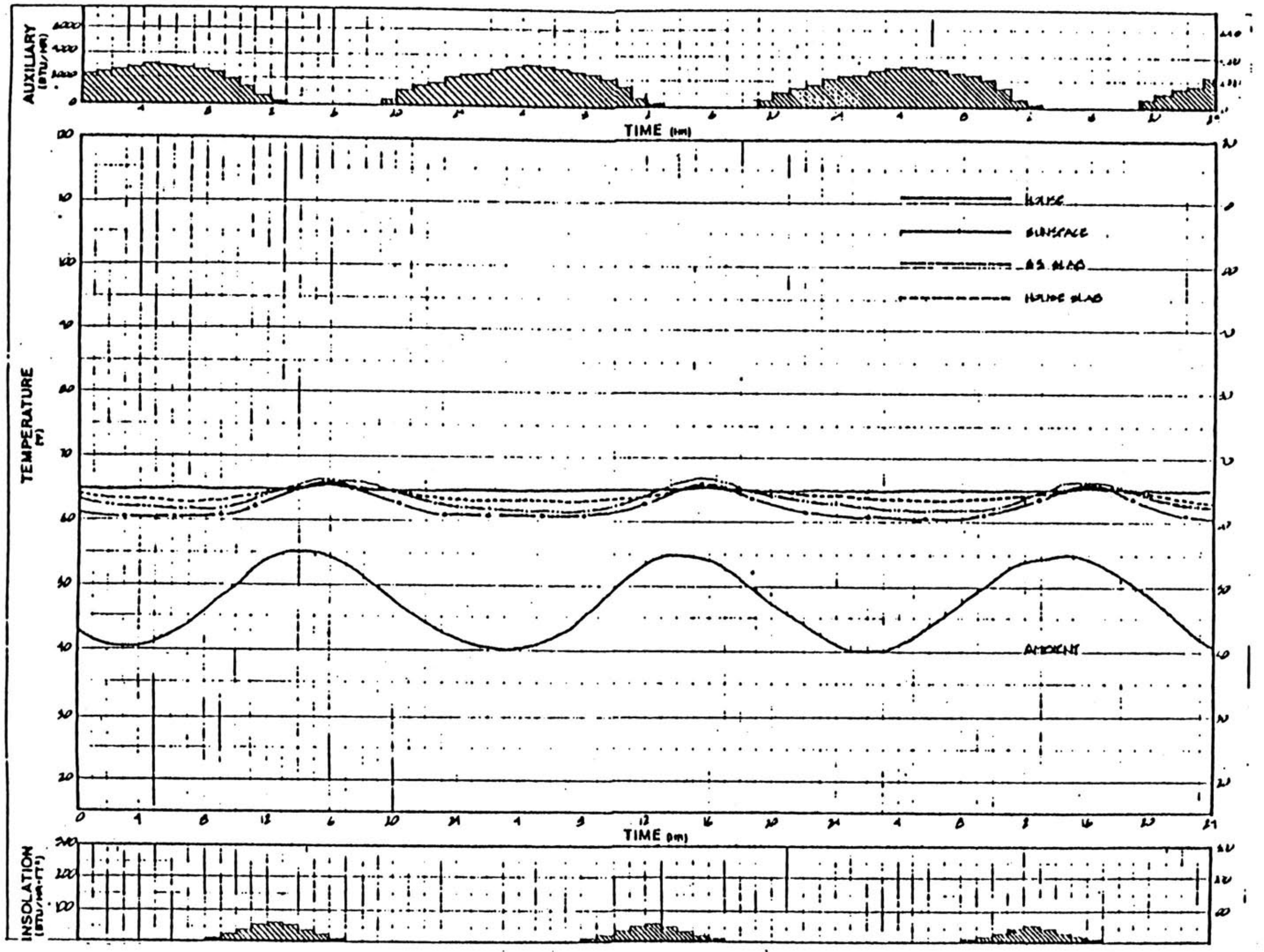
The class, while having many constructive comments about the design tools we used, had some negative ones as well. Everyone wished we could have had access to CALPAS3 all through the design development phase, and not just at the working drawing stage. With all it's limitations, TEANET was not a powerful enough tool for design development purposes. At the same time, the class agreed that CALPAS3 would have been "too powerful" to have been the only design tool used because the complexity of input required was inappropriate and too cumbersome to have to deal with during the earliest stages of design. This was in spite of the fact that the CALPAS3 manual boasted about the program's ability to default most of the inputs in the case of a simple building such as a house. One found out very quickly that this didn't really make your life any easier, when you got your output back. The only way to explain your results was to go digging through many pages of information you hadn't needed to read before, in search of the particular default value that could explain your situation. And because there were so many variables defaulted, you could never explain the mess you got!

Nobody liked the fact that it was so difficult to enter data into any of these simulation programs. Data for TEANET was awkwardly prompted for a keystroke at a time, without any actual message which let the user know where he was in the input sequence, and what was supposed to be entered at that moment (the big reason why we did 25 total runs, but only ten real runs- was that half the time we lost track of where we were!) CALPAS3 on the other hand, required "batch input" which was sent to the computer all at once as opposed to TEANET's interactive input. The main problem with this was that since CALPAS's input was so complex, the user would invariably make lots of mistakes typing it in- which the computer would not catch until it had the opportunity to receive all the input. (Which due to the fact that it was a timesharing system meant that you had to wait overnight just to find out you made a typo!) In any event, the computer had problems telling you exactly where your errors were- since by this method it could only guess itself- in exactly the same way that a compiler has to. And it was obvious that CALPAS3 was not as good as a compiler at parsing out text! What made CALPAS3 so hard to use, was that errors in the beginning of the input file would invariably make the program come back with many erroneous error messages because of this problem. Of course in a strict sense, the program was always right- but unmercifully, due to the strict syntax this method of input required. TEANET on the other hand, suffered greatly from lack of anything but a poorly designed interactive interface. It greatly needed some form of batch input process as well.

(If I say anymore, the reader will have no trouble guessing who the poor student was who ended up resetting his alarm clock every half hour to feed TEANET another set of input!)

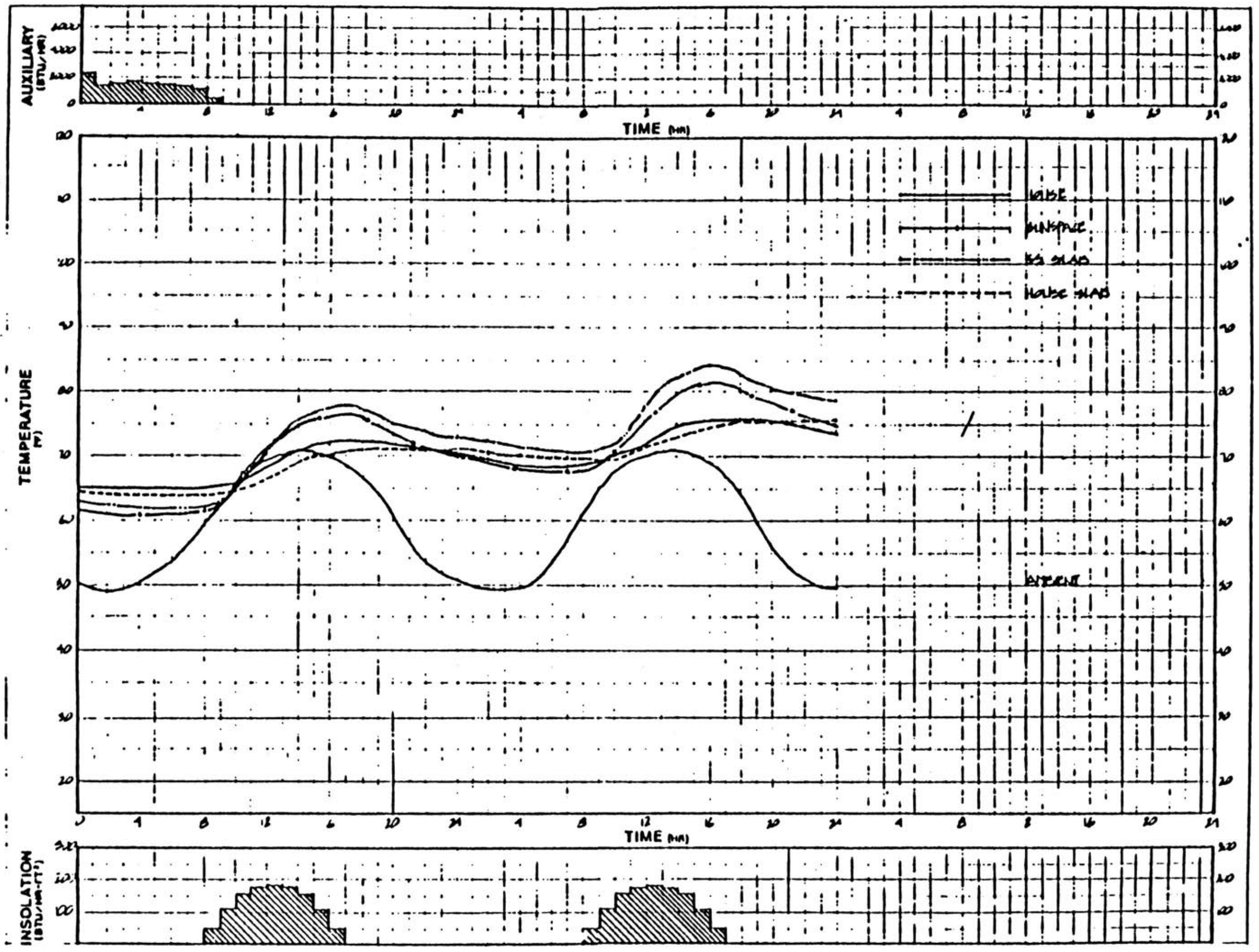
On the output end of things, I hope the reader sees in Figs. 1 and 2, how easy it is to get a grasp on how this building was performing at different time periods via the nicely drawn graphs. Let it be known that two other students spent hours slaving over plotting points from raw data that these programs put out. No doubt wishing while they were doing this that the programs had been smart enough to do it themselves. No doubt with comments like "isn't this what computers are supposed to be good for anyway ?" It was not beyond our level of intelligence to realize that if CALPAS3 could put a table of numbers on the CRT in front of us, that it could just as easily plot the dots on there itself- and a heck of alot faster than we were doing! It would have been alot easier to do away with the task of graphing all the output, but we found out quickly that it doesn't take long to get mesmerized when you are staring at a scrunched table of numbers with FORTRAN variable name column labels- such as CALPAS3 puts out. Variable names limited to three characters in length I might add. Since the names have been stripped of all their vowels in order to save space and are indiscernible, you have to resort to comparing columns of numbers with each other in order to try to identify what they are. And before you can get it all figured out, you get mesmerized and forget for a moment what you were doing. You can shift your eyes for a moment to avoid this outcome- but then you've lost your place! Either way, you end up having to start all over again. Needless to say, we didn't appreciate this feature very much, and found that this kind of distraction greatly inhibited our learning process. Graphs are the only good way to get anything worthwhile out of output data anyway, because what you need to look at are trends over time. The best thing about the TEANET program, was that the manual had an excellent chapter on how to interpret the output using graphs. As a matter of fact, we liked their method so much that we even copied their graph. It's beyond the immediate goal of this discussion to go into the details of this science, only to say that we learned that graphed output immensely increases your awareness of how a building is actually performing. It was an absolute necessity, and because of this, we were very disappointed that these design tools didn't incorporate such a feature.

In between input and output is runtime! Our big problem here was that our main calculation tool was a TI-59 calculator. Many people in the class had never seen a programmable calculator before, and were immediately fascinated by it. On the surface it seemed like such an ingenious, powerful little device. Since we only had one, and many design teams- it obviously had to be shared amongst many eager users. This posed a problem the first few weeks, but later on in the quarter, I was able to take it home for a full week without anybody saying a word- not even a peep! By that



WITH CORRECTED SOLAR INPUT

GROUP 4 APRIL- 3 CLOUDY DAYS 3 DAY THERMAL ANALYSIS (TEANET)



GROUP 4 APRIL - 2 CLEAR DAYS 3 DAY THERMAL ANALYSIS (TEANET)

time, everybody else had run out of patience with it because each run took 25 minutes. Before you knew it, hours of precious time was gone! Since nobody else wanted to use it anymore, I figured I could get away with taking it home where I could supervise it while doing other things- which is the only way I could conceive of the thing being productively useful myself.

I could not help but ponder how the status of a TI-59 had fallen so rapidly in just a couple of years. Since I had owned one myself when they first came out (but promptly unloaded it when one day someone showed me their HP-41c). And now, who would go out and buy an IBM-PC ? Just the uninformed for the most part. It was obvious to me at the time, that there necessarily had to be a good match between the problem to be solved, and the capability of the hardware and software to do it in an efficient manner, or the public simply was not going to be patient with it for very long.

All the design tools we used had serious problems in this area. The TI-59 although it could easily handle the math, could not crank through it fast enough, and by the time the HP-41c came along- neither could it. Although this machine was nearly twice as fast, in the meantime, designer's expectations had also risen in terms of what they wanted to do with the calculator- i.e.- run bigger programs, so the turnaround time for both calculators (even though one was twice as fast as the other) was roughly on the order of 25 minutes. Which although these machines were obviously cranking as fast as they could, was longer than expected for a rather simple schematic or design development type of calculation. CALPAS3 proved to be no better in this regard, even though it's CPU time which was listed on the output, was on the order of a few seconds. Since the user interface was so awkward on both the input and output side of things, the actual turnaround time was no better than with the programmable calculators. In fact, because of the much larger discrepancy between the amount of time and effort you had to put in as compared to what the computer had to (say for arguments sake- an hour as vs. four seconds!), it left many of us in a more frustrated state than the programmable calculators had. It all goes to show that psychology plays a significant role, which can not be overlooked in the development of a design tool. If it can't keep up with the speed of technological inflation, it won't be on the winning side.

As the reader might expect, all of this experience though much of it was not real pleasant at the time, proved to be much food for thought, and still is many years later. It gave me a real appreciation for computer design tools, although I learned that a great deal of engineering wisdom was needed to use them- that they couldn't simply be treated as "black boxes", or GIGO (garbage in- garbage out) would result. My overriding feeling about them though, was that the present state of design tools was very immature, and left alot to be desired. However, this rather disappointing reality gave me my first general hypothesis: that if certain aspects of the computer were designed better so as to equip it to effectively handle parametric analysis- i.e.- make it more educational in nature, in a manner which matched it to an appropriate stage of the design process, that the computer would become a much more effective design tool, and an invaluable one for architectural engineering purposes. I also learned that as a matter of course, graphical output was not only a convenience, but an absolute must in terms of being able to see the output results in such a way that enables one to get a clear picture of what is going on.

Although the fire was to die out, several bright coals were left burning, and it was not too long at all before these sparks ignited a vision on my part, to make a plunge of my own.

CHAPTER 1: FRUSTRATIONS AND DEAD ENDS

"You fool! That which you sow does not come to life unless it dies; and that which you sow, you do not sow the body which is to be, but the bare grain, perhaps of wheat or of something else."

"Tunnel vision" always seems to crop up in the absolutely worst moment. Alot is usually at stake- and was, this particular day in august on the Dead River in a remote part of Maine. It was the biggest race of the year in this part of the country- something akin to the New York Marathon, or the annual "Klondike Derby" (Nome-to-Anchorage dogsled race), depending on which side of the country you're from. I was in the bow, and my partner was in the stern. Somewhere in the middle of twenty-two miles of "wildwater" as they call it, half dead as I was by then, I was startled out of my trance by a loud shriek behind me. Faintly recognizing it above the roar of the river- as my name!, I instinctively stopped paddling and through waterlogged eyes stinging from sweat, gazed farther ahead. But straining through the spray and mist, I could see nothing to get alarmed about-even though the shrieks had turned to war-hoops by now. Not a rapid in site- nothing in site, clean!, and as far as I could see, I had picked a good course down the river. As I reflected on the fact that there must be something I was not seeing- the possibility finally hit me, and as I shifted my eyes and finally caught it- I was so stunged that I froze. Dead in front of me, barely thirty feet away- was a barely perceivable, fifty-foot wide sharp horizontal line on the water!

As the war-hoops had changed to barking by now, I figured I'd better try doing something, so I started back-watering like crazy. A second later as we started veering to the side, discovering that my partner was doing the exact opposite, I came back to my senses- realizing that we had no time to fight a big river. If you do get yourself caught by "tunnel vision"- the first thing you've got to do is to get a good grip on where you are. The second thing you've got to do immediately after that- once you've decided what the most graceful alternative is (knowing that there won't be much grace!)- is to make a fast move. If you're going to go over a waterfall- you're going to go, there's no two ways about it- so the best thing you can do is to pick up some speed and go sailing over. Unfortunately because of my misdirected reactions- we didn't hit the edge fast enough to avoid the "hole" of turbulence- nothing short of a "black hole" on the other side. It'll suck up a canoe in a split second, and it was all I could do hanging on to the gunnels with all my might- not to become a permanent anchor in the bottom of the river!

Many years later I was to remember this episode- walking up the street from the UW's academic computer center in about the same condition as I was in at the end of the Dead River Race. Very slowly! Feeling very exhausted, and not knowing whether it came from the strain of the exertion involved, or the stress of not really being able to understand what had happened. Sad that things hadn't come to an ideal end, but relieved that it was finally all over. After the passive solar class was over, I signed up for a full quarter's worth of independent study with Joel. My purpose was to round out my academic education with a real hands-on computer project, exploring the possibility of developing a better user-interface for passive solar design software. Joel got a copy of the original CALPAS1 (1) from Harvey Bryan at MIT. It was written in FORTRAN for a mainframe computer system, and I was going to try to put it on a microcomputer for the school, substituting it's batch interface for an interactive one. All in one quarter.

It turned out to be a very exciting challenge for me. I remembered a smattering of FORTRAN from an introductory programming course a few years earlier. Which was of course back in the "Middle-Ages" when everything was input with a keypunch. I knew absolutely nothing about interactive programming, had never written a complete working program, and had heard nothing about the new science of "structured programming" that was evolving at this time. But putting PASCALC onto the HP-41c had been a fairly large undertaking, so I felt I was prepared for this next step. Curious as to why so many people were talking about this new language called

"Pascal", and remembering that FORTRAN had been a very "picky" language, I picked up several books on Pascal, since our Terak had a Pascal compiler. One of these books in particular- an advanced book on Pascal which to this day is still the best book on structured programming I know of (2), got me thoroughly enthused on the subject and I couldn't put it down.

My scheme, and at that time I had no idea that anybody had ever experimented with it much before, was to develop a screen-oriented method of input based on something called "direct-cursor-addressing". Hidden way back in the UCSD Pascal manual were a few commands which allowed you to move the cursor to any point on the screen, or to get from the operating system the exact location of the cursor at any given moment. I wanted to take advantage of this ability to develop a "user-friendly" interactive interface, which had many features which were far advanced over the more usual line-oriented, question mark by question mark type of arrangement that most programs offered at that time. Screen images would be easily filled-in templates where the cursor itself would prompt you for input, check your entry against a range of allowable default values, then act accordingly. It would either accept your answer- or give you an error message telling you why your answer was unacceptable, back up, erase your answer, and wait for a new one. Help messages could be asked for at any point, and the carriage-return could be hit repeatedly if you were in a hurry and just wanted to accept the program defaults.

The day after I bought my eight-inch diskettes, the Terak died. After discovering that an Apple II with a forty column screen was useless, I was forced to move to a mini environment, and spent the rest of the quarter working on a Vax-11. I slaved away for weeks, following my books on structured programming to the letter. The key concept which they all talked about was something called "top-down design", which basically was an organizational concept that was supposed to aid in writing long and complex code. It's main theme was to always work from "big to small", putting off smaller needs until they absolutely had to be dealt with. This kept you from getting muddled in trivia, always focusing you on your largest goals so that you wouldn't lose sight of where you were, and where you were going. In addition, it incorporated a rather ingenious method of debugging where you would continually check that your program was logically correct by substituting these- as yet uncoded low-level routines with fake procedure calls. The idea, was that you would always know that the major part of your code was working at any given moment, and you would then only have to debug the small piece that you most recently added. Once you finally added all the remaining small pieces in a similar fashion- your program was almost guaranteed to work with very little difficulty. Based on my previous experience with the unmodularity of FORTRAN, this sounded all too good to be true! I worked on it day in and day out checking out all the procedures thoroughly, commenting the code, adding help facilities, etc .... Since I didn't have any low-level routines yet, and therefore the program was not completely functional, I faked the input as well. Initializing arrays of chars- just as though they had been filled by a low-level routine, I was able to essentially run the program.

The program made extensive use of the terminal ascii escape code sequences in order to do all it's fancy cursor footwork. One day, after a final assembly of some 700 lines or so of code, I was ready to try it out. To my astonishment, although everything had worked up until then- nothing happened! The terminal was dead- so it seemed. Then I discovered that it would take in characters one at a time as long as I hit the carriage-return in between each character. That was ridiculous! After many hours of manual searching and talking to the computer center consultants, it finally came out that VAX-11 Pascal had absolutely no facility to get a character from a terminal without first entering a carriage-return to signal the computer that you had something to send it. My whole scheme was ruined! Aside from rewriting the whole program in VAX-11 FORTRAN which had this facility, or redesigning all the algorithyms, my consultants told me that there wasn't much I could do. One consultant knew how to call FORTRAN from Pascal, and tried to rig me up with a FORTRAN "patch", but contrary to the VAX manuals- it never did work, and they finally concluded that certain inconsistencies between how the two languages made parameter calls would not allow it to work properly.

Two months of hard work completely down the drain! And my first hard lessons about language extensions and portability problems. That one version of a language was not the same as another (UCSD Pascal had allowed this facility), and that one computer word was not the same as someone else's computer word. So much for "top-down design" as well! There were obviously needs at a low level which needed to be dealt with early on in the software development process, which couldn't wait until last. So one day as I realized it was all useless, and finding it pretty incredulous that I had so thoroughly tested everything but the most obvious, I had recollections of similar past failures due to "tunnel vision" as I walked slowly up the hill.

Feeling really depressed, I went into a restaurant and ordered a bowl of chili. No sooner had it arrived than a man named Larry Palmiter walked in the door. Recognizing that he had seen me before somewhere, he sat down on the empty seat across from me. (If the reader doesn't know this name, Larry Palmiter is in my opinion, the guru of the passive solar design tool world. I don't what his IQ is, but at passive solar conferences, you can't stop Larry from talking on and on because quite frankly, most people can't comprehend what he's talking about, and therefore can't shut him up. He's been respondsible for such things as evaluating all the available passive solar design tools for SERI (3), and has even been able to do dastardly things to the government- like find bugs in DOE-2! (4))

Since I didn't want him to ask me first, I asked him how things were going, and as he unveiled his most recent sob story to me- I perked up. "Problems in our new program at ECOTOPE- to the point where a month ago, after nine months of work and 11,000 lines of FORTRAN, I almost tossed it in the wastebasket!" I couldn't believe it! He ordered a full-course lunch, and I just sat there for the next four hours listening to him talk about energy software design. Everything from the pros and cons of the various heat transfer algorithyms, to talk about various languages, methods of user-interface, the next generation of computers and energy software- you name it. As usual, he had so much to say on the subject, that I never got around to telling him about my problems, but they were piddley compared to his. He swore that the next time he wrote a program, it would be a "program generator and not just a program". "Because these programs are now getting so big and complicated, that when someone finds a bug that you've got to fix- it's so deeply embedded in the code that you can't possibly do it!" "A what?- that's because you're using FORTAN!"

And on it went. As I finally left at dusk I realized that I had learned as much in the last four hours as I had in the previous year about design tools and energy software. So much for formal education! And in general, Larry and I had agreed on almost everything in principle, on where the field was, what was most lacking in energy software, and what the roadblocks were. That the primary constraint for the moment, was not in the algorithyms, nor in the user interfaces, nor in the hardware (sixteen bits had just become the new norm), but in the development software, and what it didn't allow you to do. The man sitting across from me who had spent most of his years pouring over various algorithyms, was now most concerned about finding better ways to program, and programming tools which really worked! It was a real about-face, and needless to say, I was very impressed. As usual, Larry had already done quite a bit of thinking on this new subject, and through the course of the afternoon, he had energized my mind to a new level- from "tunnel vision" to new visions which- little did I know that night, would carry me 3,000 miles away to MIT!

When I arrived at MIT a year and a half later, I found out that two people in particular- Charles St. Clair and Constantine Seremetis, had been hard at work developing a screen-oriented user interface just as I had, only with a considerable amount of better succcess. Which was a real relief to me because quite frankly, I was burned out on the subject by then and was glad to be able to move on to other things. Like better languages and better methods of programming. The work they had accomplished although it was not quite finished yet, was very similar to what I had envisioned myself- except their screen images were more complicated than mine had been. The main program that they were interfacing to called Micropass (5), was a later version of CALPAS1 which had been revamped for use on microcomputers. More features had been added to it along the way, which necessitated more input and therefore more complex screen images. I really liked the work these two guys had done, but was really appalled by one thing- they wrote it in BASIC! And much to my horror, I found out that everything that had been written for microcomputers in the architecture school up to that point, had been written in BASIC. Of course, that had been the easiest and the cheapest thing to do, since BASIC came free with every purchased microcomputer. They also argued that at that time, there weren't many high-level language compilers on the market for PC's, and most PC user's didn't even know what one was- let alone own one. And if they did own one, the likelyhood that two PC owners would have the same compiler was as good as remote, since compilers for PC's was not the norm. Although Pascal was definitely the standard for a high-level language at that time, I myself had learned the hard way that no two versions of Pascal were alike in some of the most fundamental ways.

Although I couldn't buy at all the idea that BASIC was going to remain strong in the microcomputer world for very long, there wasn't much I could do except to bite the bullet, and again, go along with the way the river was flowing. I did spend some time on my own that fall searching around for language compilers. I bought a copy of Niklaus Wirth's new book on MODULA-2 (6), an upgraded version of Pascal which promised to cure all it's evils, and a special issue of BYTE magazine which was devoted to espousing the pros and cons of a new language called "C" (7), but at that time, none of the experts could agree on what language was going to become the new standard production language superceding FORTRAN, so it was really too early to choose.

The final blow for BASIC came when during that fall, I tried to convert a fairly sophisticated HVAC program (8), from Apple BASIC to IBM BASIC for the Designer's Software Exchange (9), a non-profit, public-domain software distribution center which I helped organize with Professor Harvey Bryan at MIT. Talk about a hopeless task! One reason I am sure why my predecessors at MIT were not as vehemently outspoken against unmodular languages as I was- was the fact that they hadn't yet experienced the virtues of a modular language, and didn't really understand the philosophy of structured programming. Later experience seemed to bear this theory out. That winter, after much initial resistance, I managed to convince one of these people to take a C-programming course with me, and consequently- I have never seen a faster conversion! But by this time this guy was really ripe. Because both of us spent endless hours struggling with our respective BASIC programs that fall- and really to no avail. After weeks and weeks of head-pounding, mine eventually got finished which by that time I considered to be nothing short of a miracle.

Since BASIC does not allow a programmer to declare his variables at the beginning of a program except for those that are dimensioned as arrays, new variables get made up along the way in the moment they are needed. It is common practice to give a variable a name which is a close derivation of others it is associated with, because the linear arrangement of code gives you virtually no other way to keep track of things otherwise. The result is that after more than a page of code (about all that BASIC was originally intended to do as a beginner's language), it's about as easy to read as chinese: Crammed together, everything looks almost the same- but not quite, and you can't tell whether you should be reading forward or backward! A hopeless confusion where it is impossible to find all of your coding errors without extensive use of a cross-referencing program. And as Larry Palmiter said of FORTRAN which is similarly structured, "bugs are so deeply embedded in the code that you can't get them out", and "after a certain point, it's just impossible to add any new program features!" If the reader thinks I am trying to stretch their imagination too far, let it be known that the program that this other student was working on (which will remain anonymous), even though it was of such high quality as to be presented at an annual national conference- to this day remains unfinished, and I seriously doubt if it ever will be. After a certain point had been reached, this programmer found himself painted into a corner by the computer language, literally unable to move!

At the end of that whole experience, many of us said good-bye to BASIC forever. Another set of factors however, had been influencing us toward this end in a somewhat parallel fashion. Around six months after it's inception, the Designer's Software Exchange became successful, as software from outside sources began to role in. In all, we collected about a dozen programs of various sorts which were deemed to be in good enough shape to offer in The Exchange- which without exception, were all written in BASIC. Most of the software that was submitted simply was not usable, primarily because of two reasons. It usually had a very inferior user-interface, on the order of the question mark after question mark type. And quite honestly, it usually didn't accomplish very much. After using it you felt let down because although the programmer may have had a great idea, the software fell way short of actually accomplishing it, and it left you with a piece of what you thought you were going to get- and not the whole thing. Consequently, you weren't too excited about it after all. The reason for this of course, was that BASIC lacked the power to do anything without getting inherently convoluted, so most programmer/designers- even if they had the experience to do more, simply used their PC's as an oversized calculator and not much more.

On the other extreme were the programmer/designers who tried tackling a rhinoceros- and unlike my friend, didn't get stomped on! These programs were so grossly complex that I dreaded hearing the phone ring for fear that someone wanted some technical information on them. I could never begin to unravel them enough to be able to give an intelligent answer. Even the HVAC program I wrote, 3 months after the fact, I could not answer a simgle technical question on. People would answer the phone and say, "you'll be happy to know that the author of that program is right here, just wait a minute" and I would cringe inside, knowing that just like the last time, I would not be able to answer their question. BASIC in essence, became the "Achille's heal" of The Designer's Software Exchange. When it came time to choose available application programs as "modules" to be used in the work for this thesis some six months later, all of the likely candidates except one in the BASIC catagory were thrown out because of the infeasibility of translating them, even though some of them were perhaps better programs than the ones that were eventually chosen.

During the same time period, the School of Architecture and Urban Planning established a computer lab where microcomputer facilities existed for student use. Through the software exchange and other sources, the lab began to accumulate various pieces of architectural software. Some of this was your typical canned business software, and the rest was of the homegrown BASIC variety. And pretty soon the students were inundated with twelve varieties of user-interfaces, six kinds of file usage, and more different menu items to choose from than the best restaurant in downtown Boston! It was apparent to some of us that some drastic cleaning-up needed to take place because this clutter made it impossible for those who were not daily users of the lab to be able to get anything accomplished there. From one week to the next, it was not possible to be able to remember one user-interface from the next. And with six different methods of file storage, you simply had to know twelve user-interfaces because there was no way to pass files between applications. Consequently, as a student you were forced to either use the lab all the time- so that you remembered how to use it the next time, or not to use it at all. For some students like myself who's coursework did not require us to use the facilities on a regular basis, the only alternative- since we couldn't be there often enough, was not to use it at all.

In summary, my journey thus far gave me more insights in the area of what not to do, and what not to repeat again, than it did positive and concrete findings which I could move forward with. But knowing what not to do, by a process of elimination can many times lead one to positive insights as to what one therefore ought to do instead, and this was very much the case here. My big failure with "top-down design", having been caught by "tunnel vision", taught me that my thinking needed to be very broad in scope, much broader than most people were thinking at the time. My subsequent talks with Larry Palmiter succeeded in cementing this idea in my mind. That my earlier hypothesis had been correct: indeed, there was alot to be done, and nobody seemed to have a viable solution yet. That the main bottleneck for the moment was the fact that the current capabilities of existing software put major hurdles in the way of the design tool designer, effectively thwarting the outcome of his gallant endeavors. That what to code was not the major issue, but how to code. And consequently newer, more flexible and powerful software development tools were required before existing design tool inadequacies could be done away with. And this meant that it was time to kiss conventional practices good-bye, and strike out for a new frontier.

This was a very sad state of affairs to say the very least about it- but then along came hope. News about Project Athena began slowly to filter down to the masses, including the rumor that the Athena committee was "distressed" about the lack of a really good software development environment here at MIT. I was elated to hear that someone else had reached the same conclusions! And in fact, Athena meant business and not only was going to offer some courses in UNIX and C, but was looking for software developers who were interested in developing software well beyond the current limited horizons. This was very exciting news to me, because it afforded me the opportunity to branch off from past directions, and go in the direction I wanted to go. So far, all of my predecessors within the School of Architecture at MIT, had been absorbed by the question of what to code. Nobody as yet, had really touched on the issue of how to code, which I was by now all the more convinced, was the primary issue of the times. The issue which if solved, would do alot to break the current bottleneck in design tool design.

CHAPTER 2: GAINING A VISION

"Now faith is the assurance of things hoped for,
the conviction of things not seen."

The good news that Project Athena brought to my ears was that BASIC was dead! Although Athena would continue to support FORTRAN and Pascal as software development languages at MIT, BASIC would not be supported and therefore was down the tubes. Some people in my department, having nostalgic feelings about their early romances with microcomputers I think, felt alittle remorse at this- but not me! I was eager to push onward and let bygones be bygones. And since Project Athena had already chosen UNIX as it operating system of choice, my problem of having to choose a new software development language was solved. Since 85% of UNIX is written in C, it was clear in my mind that C would become the dominant applications language at MIT for years to come. Therefore it was time for me to find out more about this controversial new language. As I was soon to find out, the whole software development industry was beginning to look seriously at C as well, having been thoroughly frustrated with the more established languages at this point, just as I had been. Consequently during the last year, there has been a fair amount of press over the issue of just what is the best applications language for the scientific and engineering fields.

As a recent article points out, most of this type of discussion usually falls on deaf ears, as programmers staunchly argue in favor of what they know themselves to be "the best language"- which only turns out too often to be the one they are most familiar with. For many decades, FORTRAN has been the mainstay of the scientific & engineering fields, as it has a large repertoire of built-in math functions covering almost every conceivable need. But the popularity of FORTRAN has largely been replaced by C. Except for those situations where too great a software investment must be forsaken in terms of compatibility with older code, it seems that most current commercial programming is being carried out in C. A rather illuminating article in Computer Language magazine entitled "C instead of FORTRAN ?" looks into the software development issues behind this recent switch (1).

One of the biggest concerns in software development these days in the wake of the many recent advances in computer hardware, is software portability. Many software vendors ended up getting burned quite badly when the computer industry made the switch from 8-bit to 16-bit processors in the early part of this decade, as they discovered that the software they had written would not work on the newer machines. This occurred because low-level hardware dependent features were built into the languages that were being used, making it impossible for a program to work on a new processor without rewriting the code. Newer languages such as C and MODULA-2, avoid this problem by putting all operating system dependent features- such as calls to I/O devices, in an outside standard library which is not part of the formal language itself. When a version of a program is needed for a new microprocessor, one only needs to change the standard library to one which supports the new processor, instead of having to rewrite the code. In an era when new processors seem to pop up overnight, this language feature is a must for software house survival. Although the standard library varies somewhat from one compiler to the next, it is usually not hard to make the relatively few adaptations required to port a progam to another machine.

Truly modular languages offer other crucial advantages besides portability and compatibility. Because they support the ability to reference external chunks of code residing in other files (i.e.- libraries), it it now possible to divvy up a large programming task amongst a team of programmers who can then go off to their own computers to work on their respective part. Team-programming was not as easy to do with the older non-modular languages which required a program to be one continuous source file, to say the least! Nowadays, when competition for user-friendliness and performance has forced virtually all software development ventures to be highly sophisticated in nature, team-programming is a must, and thus the popularity of C has risen dramatically in the last few years.

Speaking of performance, C has a definite edge over other application-oriented languages for a number of reasons. In terms of code size, C is a very compact language making it an ideal language for use on microcomputers where storage space-both in terms of memory and disk space, is at a premium. A C compiler is a very lean machine, simply because the extra baggage that the all-inclusive, non-modular compilers needed has been done away with. Executable code therefore only includes what is essentially needed in order to get the job done, and nothing more. This contrasts markedly with a heavy-weight language such as FORTRAN. Write a five line program in FORTRAN and to your horror you will discover that the executable code takes up about 30k! From my own experience, a similar program compiled in C would be under 10k. In terms of execution speed, C offers two distinct advantages over most other number-crunching languages. C supports many low-level operators which mimic the assembly language instructions that microprocessors use. This reduces the number of clock cycles required to execute a sequence of instructions resulting in faster code (2). The other advantage which has become of paramount importance now that the public is clamoring for things such as pull-down menus, viewports, and other screen intensive functions, is the ability to directly interface with the assembly language that the operating system uses to control the whole computer (3). These features would be much too slow without the use of assembly code. But the fact of the matter is that they can't be done without it anyway. In the past, the only alternative was to write the whole program in assembly code in order to provide these features. This made the code completely unportable, which is where alot of software companies got burned in the switch to 16-bit processors. Realizing that the corresponding switch to 32-bit processors is inevitable once the industry is pressured to the point, many of these companies are now writing the majority of their code in C, keeping what assembly code is absolutely necessary off in separate library modules which can then be easily replaced in the future.

Contrary to what most people might believe, software maintenance costs take up roughly fifty percent of the total software budget (4). Thus the main emphasis in software development projects in recent years, has been to try to write code which is "packaged" in a compartmentalized fashion so that any programmer (not just the original author), can pick it up somewhere down the road and easily make changes to it. Thus the emphasis these days on structured programming and so-called "top-down design". These programming style techniques which make programs easier to understand, are for the sake of programmer productivity. Consequently, the odds are good that a new programmer can pick up a program that has been written in a procedure-oriented language such as Pascal, and quickly see what has to be done in order to maintain it. But other than to save the programmer some time twiddling his thumbs, it does not necessarily make his task that much easier, because changing code usually involves to some extent, changing the underlying data structures.

One big advantage of C due to it's ability to address both data and blocks of code in memory via pointers, is that it affords the programmer the opportunity to write segments of code which are "generic" in that the underlying algorithym becomes independent of the particular data structures used (5). If in the future it becomes necessary to change the form of the data the program uses, the program can then be made to "point" to new data structures, and the corresponding new pieces of code which manipulate them. For example, it is entirely possible to write a program which sorts data where the program really couldn't care less what it sorts (6). One day it might sort integers, the next day it might be called on to sort double precision floating-point numbers, or even something considerably more exotic, using essentially the same sorting algorithym. This ability, in conjunction with the facility to create libraries of source code means that the programmer has the opportunity to write code where large segments of it can be "plugged-in" and "plugged-out" on relatively short notice to accomodate some new and unexpected need which has arisen.

Due to the fact that C offered so much power and at the same time, so much flexibility, it quickly became evident to me after finding out how well accepted C has become in the scientific and engineering fields, that C was the language of choice for any near-future architectural applications. That as a software development tool, it offers the programmer more hope than any other applications language does at this time, that something started may even get finished, and won't have to be scrapped later on for lack of compatibility with newer hardware and software products, or because nobody can maintain it. Problems which had fatally put to death all previous attempts by my colleagues in the architecture school to produce viable in-house software. Moreover, UNIX's symbiotic relationship with C meant that students would be encouraged to learn it, and therefore two or three years after I left MIT, someone within the department would most likely still be interested in maintaining any software I developed in C, as a means of initially aquainting themselves with the language.

With all of this new learning on the tip of my mind, I was only too eager to take Project Athena up on it's C-programming course offering. The best aspect of the C-programming course was not the fact that I learned the language. The most important thing C did for me as I started experimenting with it's power, was that it began to open up my mind to a new awareness. An awareness that showed me that I shouldn't be inhibited by the frustrations of the past- but that I should begin to think on an entirely new level where you didn't need to set artificial limits on what you thought was possible. The most far-reaching aspect of this new awareness, was that it was time to stop thinking about individual programs. Experience had shown me that in the long run, individual programs didn't accomplish a whole lot because they were in effect, like little isolated islands in a huge ocean. In many respects, this adequately describes the state of affairs in the architectural software world, and how it came to be. Lots of architects off on their own little islands, doing their own thing on a different computer. Not much chance that any of them would find much use for what their counterpart somewhere else had been doing (once the communications link between islands got invented). C brought about the possibility given a new perspective from the start- i.e. a more global perspective, that individual applications could be written not as individual programs, but as "modules" which could then be plugged into a larger framework which supplied the necessary communications and support facilities to allow these applications to function together. The result would be a synergistic system where the sum would definitely be greater than the sum of the individual parts.

For example, it would then be possible to try to optimize the design of a building based on more than one criteria, which- due to the isolation of programs, has been the norm in the past for building technology software. Every architect knows that you simply can not design a building around a single set of criteria because at some point this will conflict with another important need. A case in point where such a "software system" would be valuable in the design process would be something like the following, which is not all too uncommon these days: An architect designs a commercial building based on an open-plan scheme so as to be able to use daylighting to cut down on air-conditioning loads. An energy analysis program is employed in order to optimize the design of special design features which enable the building to maximize it's use of daylighting. Once the building gets built and the occupants move in, all of a sudden they find out there is a big acoustical problem because there are no internal walls or partitions. So they add some. Now the daylighting scheme no longer works adequately enough because the space has been chopped up, and they now are forced to turn on much more artificial lighting. Consequently the air-conditioning load jumps up (during peak hours of course) accordingly. Since the building was originally designed with special daylighting features, you can bet that it's cost per square foot is greater than a "normal building" which utilizes artificial lighting as the primary light source. At this point, the economical advantage behind using daylighting has been lost, since the owners will not be able to recover it's additional capital cost.

A situation such as this could have been avoided if from the start, the software employed had the ability to explore the trade-offs between different building system requirements, and somehow was able to keep track of the marginal cost associated with each design scheme as it was being simulated. The results could be depicted graphically on a CRT, giving the designer a quick and easy means for making a rational judgement as to what scheme produced the best results according to the criteria chosen. Also, by making many runs while changing only one or two building parameters, "sensitivity studies" could be udertaken to try to determine what building system elements were most critical, and thus should receive the most attention in this optimization process.

Such a software system composed of integrated program "modules", would obviously require certain features which from an initial conceptual standpoint, seemed quite doable given the facilities of the C language as I have previously described. These modules would need the ability to function independently as separate islands within the larger context of a whole system of design tools, yet be able to communicate with each other in such a fashion that they could be used together as problem solving tools. This of course means that they must be able to share their input and output data so as to be able to benefit from each others analyses.

On the input side of things, since there would invariably be alot of unnecessary (and time consuming!) redundancy if each module was solely respondsible for getting it's own data from the user, either a common "universal" user-interface was needed which could accept all the information pertinent to all modules, or some means of sharing input data- i.e.- conserving data from one module to the next was required. There would of course be many inconsistencies amongst the datasets of the modules. Although modules might require the same basic data about a building, they might require this data in a slightly different format from each other, and this would pose compatibility problems which would have to be dealt with.

On the output side, all output from any given module must reside in a form which is compatible with that of all other modules- otherwise there would be no means of comparing the results of one module's simulation with another. Since the user could, and inevitably would in order to take full advantage of having such a system- be making a wide range of parametric studies on a project, all of which would not be made at the same time, some method of archiving module output data must exist so that it could be recalled at any later time from some kind of database storage device. Thus the particular output data format chosen would have to be compatible with this facility as well. Although this output data would probably be most naturally displayed in typical tabular form, my previous experiences reminded me that it must easily be amenable to graphical display formats as well.

Of course the ultimate "fly in the ointment", was that there simply was no way of knowing what the data requirements of future modules would be, and therefore, whatever was developed in the meantime would have to have within it, an infinite amount of flexibility for the sake of future software maintenance. As long as this flexibility would not detrimentally degrade performance!

In terms of user-interface considerations, such a system could obviously be a nightmare to work with, and some means needed to be found which would afford consistency- both for the sake of streamlining the process so that it would be efficient enough for parametric analysis purposes, and also for the sake of the user. Since the user's mind is limited in it's capacity to remember pertinent details, a particular "style" of user-interface should be chosen and remain consistent in it's use throughout the whole software environment. Then if the user forgot a particular command for instance, at least he would have a "feel" for what to try. This would hopefully eliminate somewhat what I call the "user blues" associated with some large user-interfaces, such as elaborate text editors where it's impossible to get anything done unless you really know the system. Many of the commands are so complicated and seem so unnatural that you spend half your day hitting the "help" key!

The user-interface would for the sake of portability and maintenance, have to be detachable from the underlying algorithyms in the modules. Otherwise the same hopeless mess would result as had always resulted in BASIC and FORTRAN programs whenever changes were required in the code. Enough has been said previously about the difficulties this poses for the programmer. Getting rid of complicated screen calls by making them generic enough to be stuffed in an outside library would help this situation immensely. Of course the other reason for detaching the user-interface is that some sort of interface would have to exist at the system level, independent of any module, because there would be times when the user would need to interact with the system as a whole. This system interface would need to be consistent with all module interfaces, giving the user the opportunity to do such things as specify a batch job to be run overnight, allowing the user to peruse data files, or connect the user to other programs residing outside of the software system for subsequent data analysis or documentation purposes. Not to mention the necessary task of just plain explaining all the various system options.

The concept of an autonomous "software system" was easily conceived in my mind because UNIX allows one to define and set up his own "command shell"- i.e. the command-interface part of the operating system which remains "in control" after a program is executed. This would both from the point of view of the programmer and the user, give the system the aspect of coherence- as the software system would be able to maintain control once a particular module ended until the user explicitly told it to relinquish control back to the operating system's command-interface. Thus the user could spend time in the software system's local environment- graphing results, doing some spreadsheet analysis on some of the output data, transferring data to an external database, etc.... without having to be actively engaged in running modules.

These were some of my initial visionary thoughts when it became known that Project Athena was actively looking for software development proposals for the upcoming year. Although I didn't have a clear idea in my mind of exactly what kind of form such a system should take, I at least had a pretty good understanding I felt, of what it's major requirements were. Enough at least to be able to put these ideas and understandings together in a presentable "package" which Project Athena might buy. For the sake of presenting a clearly defined package to Athena, these concepts were given a tenative form- i.e.- the concept of a centralized "building systems data processor" for lack of a better phrase, which somehow quite miraculaously could do all these things. Of course the reader/designer will realize that this "talking off the top of our heads" was just part of the normal, creative, scheming process that we designers always go through- and in retrospect considering the eventual course of action, are usually afterwards- somewhat embarrased about!

One interesting anecdote concerns my diagram which depicts what such a system would look like on a conceptual level (See Fig. 1). At the same time, JF (Joseph Ferriera), in the Urban Planning Department was working on his proposal to Athena concerning his vision of what the Architecture School's future computer facility should look like. Although his diagram was more elaborate because it included a broader scope than just building technology software, the overall structure of his envisioned system, including how various elements were tied together was virtually the same as ours. Although there might have been a question of "which came first- the chicken or the egg", to my knowledge these virtually identical views of the ideal architectural software world were conceived in isolation from each other. It was a welcomed encouragement for me, signifying there was a chance at least that I might be on the right track. There was nothing to do in the meantime but to mull over these concepts during the summer while the Athena committee mulled over what

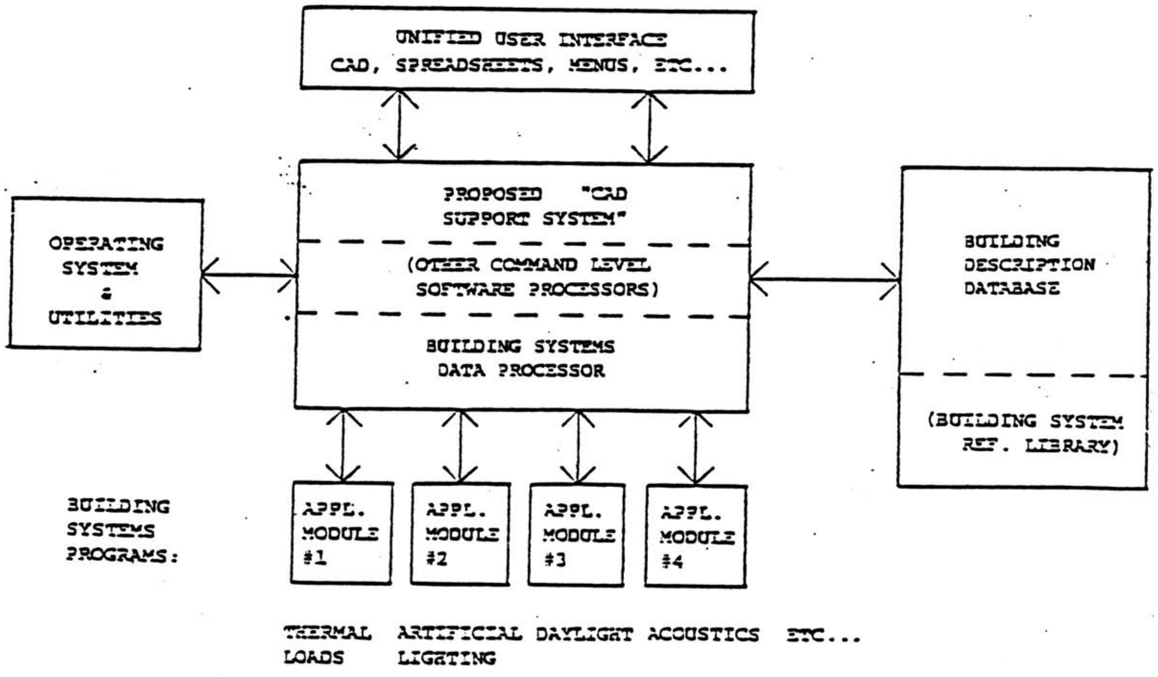

UNIFIED USER INTERFACE
CAD, SPREADSHEETS, MENUS, ETC...
PROPOSED "CAD SUPPORT SYSTEM"
(OTHER COMMAND LEVEL SOFTWARE PROCESSORS)
BUILDING SYSTEMS DATA PROCESSOR
OPERATING SYSTEM & UTILITIES
BUILDING DESCRIPTION DATABASE
(BUILDING SYSTEM REF. LIBRARY)
BUILDING SYSTEMS PROGRAMS:
APPL. MODULE #1
APPL. MODULE #2
APPL. MODULE #3
APPL. MODULE #4
THERMAL LOADS
ARTIFICIAL LIGHTING
DAYLIGHT
ACOUSTICS
ETC...

Fig. 1 The BSDP's central role in a Building Technology Software System as an integrated part of the Project Athena Architectural Workstation.

they were going to do, and try to pick up some more experience programming in C. And as a stroke of divine providence would have it, the opportunity arose to do just this.

W.A. Wright Associates is a small firm located in Lexington, MA. The man at the helm of this group, is an avid C enthusiast who was currently involved in writing an energy design tool for- shall we say for the sake of confidentiality- "a rather large an inefficient beauracracy". And as cases like this usually require, he was looking for some help. Over the course of the summer, I was to get a tremendous amount of hands-on experience writing C-code, much of which I found out later on was directly applicable to the needs of this project. But by far (and I say this using 20/20 hindsight, as at the time I could hardly appreciate it), the best aspect of this summer work experience was the opportunity to be able to write software under "real-world circumstances"- i.e.- working for "a rather large and inefficient beaurocracy" because little did I know at the time, I was going to have to deal with this later on by myself.

What ended up happening, was that the client was unable to make decisions as to what exactly it was that they really wanted, no internal walls or partitions. So they add some. Now the daylighting scheme no longer works adequately enough becaua different idea regarding what they were looking for in the end, which meant week by week, those of us who were writing code didn't really know what we were shooting at. But at the same time, business going on as usual, we had to make sure that we were doing something productive. All I can say is that our saving grace was the fact that we were programming in C. We couldn't exactly do "top-down" programming because quite frankly, we weren't sure what "top" was, but at least we could do "from the middle- up and down programming", which was a heck of a lot better than just plain "shoot'n from the hip", which would have been our only choice with a nonmodular language. So we were able to say to ourselves, "well, if we're going to do something like this in the end, then we know that we'll need something like these functions here, so if we go ahead and code them using fairly generic library routines, then they'll end up being useful in the end, whether we end up using them in the exact manner we've envisioned or not." As luck would sometimes have it, three weeks after we had started a program and were on the verge of finishing it, the phone would ring, and we'd find out that it wasn't going to be as useful as we had thought. The client was off on a new tangent. But luckily, the new tangent wouldn't be that far off from the old one, which meant most of the functions we had developed to code the program were still useful. It was just a matter of revising and reorganizing them into a new program. Not quite this simple, but a far cry from having to write something new from scratch.

One of our biggest continual hassles in this regard, was that the program we were writing required the use of some huge data files, which our client was compiling from various sources. Two of these sources we had access to fairly early on, while the format of the rest of the data we would eventually use- was as good as anybody's guess. Our overall goal was to in essence, find the common denominator between the various input files and the program data structure requirements- so that we could produce a final input file which was compatible with all these needs. But because we could never quite figure out what all these needs were, we were constantly put in the defensive position, having to say to ourselves- "what is the safest course of action, the one that is least likely to catch us in the end ?" Because I had never worked in this sort of environment before, the thought had never occurred to me that most of your time on a big project gets eaten up with these sorts of tasks and vagaries- but it's the truth! Consequently, this real-world experience of learning that coding in an academic environment is miles apart from coding in a business environment (unless of course you are coding for THEE academic environment), was a real eye-opener, and an invaluable one at that. Once you've learned this fact of life, then you've got to deal with the real problem of "how in the world am I going to get anything done if I continually get side-tracked by these off-beat tasks!" All I can say in retrospect, is that everything you do in some way, shape or form- has to count toward the whole picture in the end, or you'll never see it! Through all this, I was beginning to see that the task ahead of me in the fall was a monumental one- much bigger than I had ever realized. I was glad that I had the foresight to pad the Project Athena proposal with months of time devoted to "background research". At the time I wrote the proposal, I had no intentions of doing such a thing, I just knew that I'd better stick something in the schedule that would chew up a significant amount of time- just in case I needed it. And at this point, I was sure I would.

By the end of the summer we were both pretty fed up with "rather large and inefficient beaurocracies", having run out of patience by then, and I was eager to get on with my own work. Although I still had no firm idea really of how to go about developing a complete "software system"- on the scale that I had envisioned, at least I had become fairly proficient at writing C code, and just plain learning the art of how to get things done. And I had learned an incredible amount about how to go about developing software when the end was too far away from the beginning to be seen. I knew this was an invaluable discovery, and one which would probably end up being my saving grace- the knowledge that true faith is a virtue no programmer can be without!

But aside from growing in my capacity to be an effective programmer, by now I had gained some very positive, concrete insights into the task at hand, as I was beginning to see clearly the real-world details of the overall problem, and what it was going to take in order to solve it. This meant that I had finally arrived at my first real concrete hypotheses: That an entirely new perspective was needed where architectural application programs could no longer be thought of (and therefore coded as) independent entities, but due to the fact that architectural design is multifaceted, a systems approach was necessary if architectural software was to be of any real value for design optimization purposes. This meant that powerful and at the same time- flexible software design tools and methods were needed, and I was glad to be convinced that the C programming language could offer both. Moreover, that code organization had to achieve a new level of quality, which up until this point in time, architectural software had managed to avoid. That the issues of coherence, compatibility and flexibility- both from the standpoint of the code and it's ability to be expanded and maintained, and of the user in his various interactions with the design process, had to be dealt with in an effective way. And that these issues in an educational setting, have a far greater priority than the traditional notion of "performance". That this in turn, opened up many avenues which needed exploring in the way of coding technique, including the idea of some sort of "integration". And last but not least, due to the realities of real-world software development- investigating these issues was not going to be easy, therefore one's frame of mind must constantly be "what course of action is least likely to catch me in the end!"

CHAPTER 3: VARIOUS TRIALS

"Consider it all joy, my brethren, when you encounter various trials, knowing that the testing of your faith produces endurance."

When september finally came around, it was all too apparent to me that working for Project Athena as a software developer was definitely not going to be a gravy job. That my experiences dealing with "rather large and inefficient beaurocracies" had only begun. My first inkling of this stark reality came during the summer when Athena became quite far behind schedule in determining which projects they were going to support during the following academic year. While their original request for proposals had originally been slated for June 1st, at the last minute, this deadline was extended to July 1st. Consequently, although I had originally planned to hear back from Athena somewhere around the first of july concerning the outcome, it was nearly september when I finally got the news that our proposal had been accepted. This was of course good news, and I was delighted to hear it.

Joseph Ferriera was the man at the helm of our brig (nicknamed "the flagship"), and as a brigadier, the rest of us deck-hands soon found out that he runs a very tight ship! Having a tough commanding officer is not so bad if you happen to agree with him on the way to approach most issues. Unfortunately I think to some of us who were accustomed to the very unregimented nature of architectural design, there was all too often the threat of having to "walk the plank" looming somewhere on the horizon. At least I am speaking for myself, as one who was acutely aware that Project Athena's slowdown was going to cause me some real difficulties during the year, and although Athena wasn't going to come through on it's end of the deal (i.e.- the promised and proper support for software developers), I would be fully expected to come through on my end. All I can say in retrospect is- thank God I'm a good dinghy rower! I can usually manuever quite well in heavy water without capsizing (notwithstanding my one adventure with waterfalls), and make it safely to shore. So consequently when the inevitable finally came up which it did a few times, "well hey- if you don't want to tow the Athena line- well there it is!", I found myself madly rowing on my own.

Of course this is all a well chosen figure of speech. What I mean to imply by it, is to give the reader some idea of what it was like for me to be a part of the architecture school's flagship at times, and especially at the beginning. I'll be the first to admit that JF is a good commanding officer and there was never any question in my mind as to whether he knew how to run a flagship or not. But because he was in charge of the whole fleet, he also considered himself to be the captain of dinghies as well, and at times I seriously questioned whether he could row a dinghy to save his life! Which under the circumstances, is literally what us dinghy rowers were being asked to do. I am not trying to be facetious here. Being a captain of a flagship and being a great dinghy master are two entirely different things, requiring a different set of skills as well as a totally different mindset. When you're the captain of a flagship, you're looking down on the waves from a nice safe spot up above. When you're a dinghy master, you're right in the middle of them, and they are bigger than you are! You have to respond to them and you have to do it quickly. Consequently the perspective of a dinghy master is usually at the opposite end of the spectrum to that of a flagship commander, and your priorities in the moment are almost always completely different. My chief concern was always "given the size of the swell, how can I save my dinghy". I always had to have my bow pointed directly into the waves, or I'd ship too much water and run the risk of capsizing. I needed some good oars too, which the flagship didn't provide me with, and this made my task that much more difficult. The flagship commander wasn't concerned about waves at all. He had to get his fleet to Treasure Island within a certain time period- i.e.- nine months, consequently he spent his time wandering all over the high seas looking for it. No sooner would I get my little dinghy lined up properly into the waves, when I'd discover that the flagship had altered course and I was no longer following it. If I changed my course to follow it I would surely capsize, so there was nothing I could do but to keep heading the direction I was going and have faith that Treasure

Island was out there straight ahead somewhere. That after it had checked out all the surrounding islands, the flagship would end up somewhere down along my course. And as long as I kept on rowing hard, maintaining a straight course, we'd eventually meet up again. This is what eventually happened nine months later in most respects.

My first encounter with "walking the plank" occurred over my decision not to follow the flagship's lead regarding the choice of a software development environment for this year's software development efforts. I did not go along with the myth that some version of UNIX would be in place on the IBM-side of Athena (the architecture school was given IBM hardware by Athena instead of DEC hardware), in time for us software developers to make good use of it. From my point of view, I knew that I had so much to do that I could not afford to waste any time during the year tinkering with anything that didn't by this time look like it was a guaranteed winner. I knew that my need to rewrite code written in BASIC and FORTRAN would chew up an enormous amount of time. Consequently I could not afford the luxury of devoting precious time to learning how to use the UNIX mail facilities, editors, and other features of the UNIX operating system in the DEC environment, on a far-fetched supposition that before long Athena would have this stuff running on the IBM machines as well. If Athena wasn't prepared to offer us a fully working UNIX environment by september when our projects were required to start, then from my perspective on the water, nobody from

Athena had any business asking us software developers to prepare to port our work to a hypothetical future operating environment- given the fact that they also wanted us to have a substantial product to show at the end of the year. That was too much like wanting your cake- and eating it too!

Consequently, from day one I pretty much refused to work with UNIX and devoted my efforts toward the "MS-DOS" operating environment, which by that time had almost become a naughty word to Athena personel. The problem with this course of action was that since Athena had not intended to support MS-DOS in the beginning at all, they were not willing to invest any money to provide us software developers with a good software development environment in the interim period, during the time it would take IBM to get the IBM-side of Athena fully operative. Although we had access to the Lattice C compiler, we did not have access to any of Lattice's other software development tools such as symbolic debuggers, cross-referencers, and screen I/O libraries, which are considered to be essential tools these days by most C code writers. Knowing that translating code from a non-modular language was a formidable task, I opted to use the DeSmet C compiler with it's full compliment of software development tools which included in addition to the above, an assembler, linker, profiler, and libraries of graphics and screen I/O primatives, all of which I have used during the course of this year. I was sure that JF was not going to let me get away with it in the end (which he has recently assured me he will not!), but at least in the mean time I was confident that I could really get some work done, and consequently would have a good product to show when it came to demonstration time, which came around much sooner than I expected. Luckily, I did in fact have something substantial to show at that time, and I attribute this directly to the fact that I had a symbolic debugger which I made constant use of. In the aftermath of it all, it appears that Athena has reneged on it's committment to a UNIX-like operating environment on the IBM-side for a period of two years, due to the difficulty it has had getting the proper supporting hardware links set up (1). I'm glad that some of us dinghy rowers had enough foresight to see this coming, and spent our precious time working with MS-DOS, using our own software development tools. (It looks like Athena has decided that software development tools such as symbolic debuggers are pretty indispensable after all, as they plan to offer these tools next fall for Lattice C, along with a library of screen I/O routines.)

Not only did the flagship have a predetermined course as far as the use of a software development environment, it turned out that it had a policy regarding how code should be developed, at least to the extent of defining how existing software resources within the school should be used in order to save us dinghy rowers some effort. While I give JF full credit for the brilliancy of this idea- (I think one would surely have to be at least a three-star admiral to have thought of it in the first place), I wondered however, if it would look as good from the level of a dinghy on the water as it did from the bridge of the flagship. The idea was that as much as possible, we software developers would utilize existing canned software to perform our tasks, which would in the end save us alot of code writing. For example, we would use the spreadsheet environment as an input and output environment for our analytical programs- (which of course totally encompassed the work I was doing). The programming power afforded through the use of spreadsheet macros for instance, would enable the spreadsheet to maintain control over a series of input screens, providing a coherent user interface for students who were already accustomed to using a spreadsheet envoironment. In the same manner, the spreadsheet could be used to display program results. Output data being fed back into the spreadsheet from a calculation routine written in C, could then be tabularized and automatically graphed by the spreadsheet's integrated graphics commands. While I didn't doubt that the spreadsheet could be used for graphing output, I did doubt that it had the capacity to cope with a highly interactive process such as a user-interface. Even if it could be made to work at all through enough programmer trickery, I didn't believe that the level of performance you'd end up with would make it all worth it. I had already found out that writing a user-interface was a difficult task using an interpreted language like BASIC, and I couldn't possibly imagine that spreadsheet macros would be fast enough to do the job.

Mostly in order to try to prove to the rest of the flagship crew that it probably wasn't worth much explorative effort on our part, I did undertake the task of trying out a spreadsheet inter face for one of my modules. I ran into some hangups concerning the limits of the spreadsheet's programming capabilities, and a subsequent phone call to the software developer told me that the clean way of solving the problem was not an alternative. The dirty way left me horrified, because at certain points in the program it forced the user to watch the programmable macros jumping through all their antics. Although my experience with spreadsheets at this point was very limited, due to the slowdown in performance, I concluded that this technique was not a viable option. Months later I found out that JF wrote a program which demonstrated a queing process using spreadsheet macros which was considerably cleaner than my attempt had been. However, it took an unbelievably long time to execute. I was happy to hear this because it backed up my earlier decision that this was not a good technique from a production standpoint.

Another major aspect of the flagship's first battle plan was to rely heavily on central batteries (i.e. centralized databases) for all major operations. Mainframe relational databases running on a VAX were to be used for all file storage aside from temporary intermediate files which could reside locally on a hard disk. The purported advantage to this strategy was that many applications could have access to the same data via networking to this central file archive. Also file redundancy would be eliminated for the most part, for those applications which used essentially the same data. Of course the price to pay for all this legendary file efficiency was inefficiency in terms of the added time it took to perform all the networking and connect calls between an application and this database, not to mention the fact that the data may not at this point be in the proper format to be directly used by the application. Nobody knew what these costs would actually turn out to be since the networking hardware was not yet in place to test things out. The writing on the wall came however when Athena became very reluctant to give out database log-on accounts to software developers for fear that they would soon overload the system (2).

Although few were willing to take the risk of "walking the plank"- (they were going to have to live with Athena's flagship alot longer than I was), I think there were a few issues here which did not go over real well with those on the crew who were trained with the "architectural mindset". Primarily because of the fearful political implications that were implied by some of the Athena committee's decisions, which made us feel like we were no longer part of a small design group, but had just been deceived into joining a big corporation. The original idea behind Project Athena was that the Athena environment would end up being a network of microcomputers, microcomputers which were powerful and fully capable of handling their own concerns. Somewhere along the line this plan was altered. Whether this was due to actual hardware realities, or more to the political reality that those who had long been acquainted with large pieces of hardware such as VAXes and had alot invested in "high end" computing environments at MIT were in a very good position to heavily influence Project Athena's direction, I do not know. Somewhere however, Athena decided that micros were not going to be powerful enough to handle the institute's needs, and VAXes were needed to provide adequate CPU power for the micros which would in effect become more or less "dumb terminals". At least this is what our initial fear was, particularly when the whole issue of storing all files via a mainframe database came up. I think several of us felt that this was a wrong decision on the part of Athena, which six months of 20/20 hindsight later, is beginning to be proven true. There has been a considerable delay in the implementation of Project Athena's scheme due in part to the difficulties imposed by trying to network together "high end" computing devices with "low end" microcomputers. In the meantime, microcomputers have already jumped to the next generation. The two companies which perhaps have the most influence in the microcomputer hardware world- IBM and AT&T, have just unleashed new micros with formidible power, which are a big step forward in bringing what was previously in the catagory of the "32-bit supermicro", down to the price level which design offices and institutions can afford (3). These machines are also powerful enough to run versions of UNIX by themselves, sporting low cost 20 megabyte hard-disks. In addition, networking facilities are available to couple these "low end" machines together so that they can share files across the MS-DOS-UNIX boundary, linking together UNIX power and file-serving capability with MS-DOS's vast stronghold of applications software.

Many of us foresaw this happening when we started our Athena Projects- as well as the inevitable outcome that by the time Athena eventually does get it's current 5-year plan working, real "32-bit supermicros" linked together via some form of low-cost narrow-band networking, will have become the new norm in the business world. In effect, each black box sitting on a desk will have the full power and capabilities of a VAX, and as the war in the Falkland Islands proved to the rest of the world (where was the Athena committee ?), old heavy guns sitting on big flagships are no match at all for a squadron of fleet-footed cruise missiles! My prediction is that by the time Athena gets it's plan fully implemented- it will be from the rest of the world's standpoint, an outdated system, and they will be looking to dump their VAXes and replace their low-medium end PC's with the 32-bit atom smasher variety. If for any other reason, just to stay current with the rest of the world- which was their original plan via STRITECK cards in the first place. While rumor has it that the STRITECK cards were not found to be powerful enough to do the job, it seems ironic that in the small time period since the Athena flagship changed it's original course, these new 32-bit atom smashers have been lurking right around the corner. If someone standing on the bridge doesn't see them soon, the flagship may loose alot of headway toward it's ultimate destination!

While I respect JF's need to wear his "Athena hat" so to speak, I did not agree at all with the logic behind establishing a parasitic relationship between the software we were developing and "high end", big-gun Athena resources- any more than I wanted to develop a parasitic relationship between our software and "canned software", and I was willing to do so with my software if and only if it was proven to me that there was an acceptable match in terms of flexibility & performance, and moreover that this parasitic relationship could be broken at any given time without any painful side effects. And I did not think developing a dependency on "big guns" would be an easy relationship to get out of down the road. This is not to say that I was against using the concept of central file storage across applications via the use of database software- what I objected to was that we were in fact developing a doomed, parasitic relationship with mainframes.

The sensational success of personal microcomputers stems from the fact that they are indeed "personal" machines, and I couldn't believe that the Athena committee was oblivious to this. Did they not wonder why the "fishbowl" with it's awesome power in comparison is always half empty, while the architecture school's makeshift computer lab full of second generation PC's is always packed with students? Did they want to reinvent the intimidating psychological affects large timesharing mainframe systems had on university students during the '60s and '70s? Were we going back to the days when one had to be familiar with a whole wall-rack of manuals in order to be able to use a computer- so that the only people who do use computers are the computer jocks? If database software was to be used on the local level, I was all for it in cases where it provided a good performance match with a given application or a few closely related ones. Otherwise due to the political implications, I thought this scheme was very unwise. Come to find out, Athena has since changed it's flagship course again, and is going with a microcomputer relational database system instead of a mainframe database system.

The next topic of concern to the flagship crew was the issue of data integration. This is something that we spent weeks and weeks talking about and got nowwhere. Eventually, other issues needed to be dealt with later on in the semester and this topic was so dispersed by this point, that I think it just dissipated itself into thin air. While my original Project Athena proposal will show that I was an early proponent for tight data integration, several factors contributed to my later about-face on the issue, many of which were of a political nature similar to my distaste for "high end" resources. What happened was that we could never agree on what kinds of data should be tightly integrated together, and what kind of format some sort of "universal data structure" should take. JF was really bent on having a graphics oriented interface for nearly everything, including analytical software. We spent endless hours talking about how a universal data structure needed to be able to handle both 2-D and 3-D data, and have all the facilities to handle attributes. These were necessarily attached to points and lines on a drawing so that information from a drawing could be passed to analytical modules which could then crank out building performance analyses. This was far removed from my original concept of data integration, which had assumed only an integration of data between building systems program modules.

The problem was that there were two divergent needs here. The graphics oriented crew members were constantly referring to data as "components" because they were most concerned with the display of the parts of a drawing. Those of us who were dealing with analytical tasks were not interested in drawing parts at all, but more abstract quantities and qualities such as the gross areas of building form which had different material properties. I was conscious of the fact- due to my experience with computer simulation in the past, that it was crucial that the level of data abstraction properly match the level of abstraction inherent in the particular stage of the design process being dealt with at the moment. In this regard, the storing of data as discrete physical components definitely got in the way. The graphics oriented people did not understand this problem at all in the beginning. JF was surprised to find out that for the purposes of thermal analysis, except for the need to know which building surface faced south, the aspect of knowing the physical relationships between building systems components was for the most part completely irrelevant, due to the assumptions inherent in the modelling techniques used, which supposed a level of abstraction above that of the real world. Moreover, the graphics-oriented crew members could not even agree on what type of graphics was most appropriate for our needs- solid modelling vs. line-based, vector vs. raster display, etc...., which of course all depend on an entirely different sort of underlying data structure. After spinning our propellers for weeks over these issues, a few pragmatic realizations seemed to surface. First was the fact that trying to find a "universal data structure" was a huge undertaking which we were in no way prepared to deal with as a group. We were all talking at each other at this point, none of us having a complete enough understanding of the issue to be able to force the flagship in any given direction- much like tugboats each pushing from a different side. Eventually the pressure to have to stop talking and get some work done, seemed to force the issue to a close- thank God!

I'm not trying to say that this issue is unimportant. I think it is a very interesting one, but my practical mind told me it was also one which in big corporations would be delegated to a group of Ph. D's sitting off in some corner somewhere, who had lots of real-world experience with all the issues involved. And it would take them several years to get anywhere with it. None of us had that kind of real-world experience (as opposed to academic experience), nor did we have the time to spend getting fat heads thinking that we did, if we wanted to present something concrete to Athena at the end of the year- which is what JF wanted.

What became apparent to me though through our efforts, was that alot more was at stake than I had imagined earlier during my naive Athena proposal speculations concerning real data integration. That it was in fact one of these all or nothing situations, because the tighter things were integrated, the more disasterous the consequences would be if something were left out. What this did, was force you to try to come up with a "universal" data representation scheme, which had the potential to include everything- including quite literally from the point of view of possible architectural components- the kitchen sink! Of course it was never possible to foresee everything that might be needed in the future which easily discredited the notion that anything you came up with would in fact be "universal". And if it did work- it would inevitably end up being a colossal monster! At about the same time that these ideas were mulling around in my head, feeback via the world of PC users was beginning to be heard with reference to some of the software industries largest integrated software development efforts to date, to come up with "thee all you ever need in one business program" (4). It seems that the public is fast becoming discouraged with these heavy-weights, and would much rather see an assortment of fast, compact "cruise missiles" which can- due to their sleekness in design, really pack a wallop! A recent editorial in the well-reputed magazine INFOWORLD, discusses the recent fallout of integrated software, claiming that stand-alone applications programs continue to dominate the market (5). Reviewing the outcome of JAZZ, Lotus's new version of 1-2-3 for the Apple Macintosh, the editor has to say:

"Once again, we are reminded of the inherent weakness of integrated programs. They are like Swiss army knives: fine to have when you need a little of everything but not particularly powerful for any given task. Experienced personal computer users tend to focus on single tasks important to their work. Anyone who uses his computer principally to prepare financial analyses or to do extensive word processing needs the equivalent of either a financial jackhammer or a word processing Cuisinart, not a Swiss army knife."

When the news reached me last fall that the sales of integrated software packages was falling way behind the software industries original estimate, I began to seriously question whether in fact tight data integration was a good idea at all. Whether or not any of us were adequately prepared to deal with the issue became a side issue at this point. I began looking around for other alternatives as I remembered from my summer experiences that flexibility was my most critical need because I was trying to develop a prototypical "software system", in an operating environment which was almost certainly going to change, and most likely would even before the software was completed. That the concept of tightly integrated data- while having a certain seductive appeal on the surface, was in fact a wolf in sheep's clothing, because it's needs ran contrary to that of flexibility. It was at this point that I picked up on a phrase of Harvey's which he had remembered from Eastman's early work in architectural software development at Carnegie-Mellon University (6). The general idea was that independent programs would work together in a "cascading fashion". My attempts to dig up literature on this concept proved to no avail, nevertheless I latched on to the concept once I realized what it offered me compared to the concept of data integration. Basically what it offered was the "cruise missile" approach, where a fighter carried an assortment of firepower, each designed to work best in a particular situation. Some were directed at their target via a heat sensitive interface, while others used radar, while still others used a more sophisticated T.V.-graphics interpreter device to latch onto their target. An arsenal designed to be able to deliver the best tool for any particular situation, yet one that has the flexibility to handle those more complicated, overlapping situations. What flagship could stand up against such a diverse and powerful system!

Putting all rhetoric aside for the moment, I think a frank discussion of all the implications involved in what I have said so far is in order. It should be apparent to the reader by now, that two factions were developing amongst the flagship crew over the large issue of how to go about developing a large scale computing environment, which is a very similar problem to that which faced the early colonists 200 years ago when after winning the revolution, they were faced with the similar question- "how do you go about developing a large scale country?" In both cases, what was going to be developed, was far greater than anybody's experience, and as I see it, neither group had the ability to foretell what the consequences of their actions would be more than five or ten years later. On a large scale however, both bands of colonists could see that there were big philosophical implications, and the proverbial question in both instances seems to be "are you a Federalist, or an Anti-Federalist?" In other words, taking away the cultural context, "do you believe mostly in a big centralized system, or in a less tightly bound decentralized system?", and "which do you think in the long run, will offer the most stability, protection, and therefore usefulness for it's individual constituents?" Thus in terms of Athena resource development policy, there are those who see the decentralization of computing resources- i.e. a system of networked, but largely autonomous micros as nothing short of anarchy, while the rest of us, contemplating the stifling notion of microcomputers which are mere pawns to all powerful VAXes, would much rather say "live free or die!"

Yes in fact, when you do look at the far-reaching implications, software development being a big part of computer system development, is indeed a "wicked design problem". While no one can say with 100% certainty which course will turn out to be the best one, one is forced to either consciously or unconsciously, choose one of these two courses of action. You are either a daring pilot at heart, and believe your arsenal of small cruise missiles can outmanuever the big guns of a flagship, or you're a flagship commander at heart, feeling more comfortable with a huge steel deck under your feet, and like carrying a standard-issue, Swiss army knife in your pocket! But until the war is actually fought, who's to say who will ultimately win.

Needless to say at this point, after being gradually forced to go one way or the other in contending with all of the aforementioned, politically loaded issues, I did ultimately go with the "cruise missile" approach as my general software design philosophy, although I did use the concept of integration as a means of "bundling" data within applications modules, enabling each module to transfer it's unique set of data back and forth to disk, etc... as a self-contained "packet". In other words, I used integration where it did not conflict with my overriding goal to provide flexibility. I also later made an attempt to develop a utility program which pulled attributes from a CAD drawing program and sent them out to an applications module for analytical processing. Since the program had a very limited repertoire of attributes, I tried to design a "universal" way of packaging this data on the sending side. Of course it did require a unique decoding scheme on the receiving side which was specific to the requirements of the application. Because the data structure chosen needed to be capable of handling both types of attribute data that the drawing program could put out- i.e.- numbers and character strings, it's "universality" did require that it be "full of holes" so to speak, which was indeed inefficient. Because of the generality of this data structure however, the code itself was pretty efficient. Since right now, this utility has only been invoked for one applications module, I have no idea whether the scheme is really a practical one or not. In other words, one battle has been fought, but the war is not over.

As the reader is no doubt beginning to see, all of these issues gradually pushed me in a certain direction, which turned out to be the same direction I found myself being pushed during the summer. If the inability of our flagship crew to agree on anything specific was a "telltale" sign of something, it certainly had to be the fact that software written for an educational setting must be able to cope with constant change- i.e. "points of tangency", or else in a very short time it will be absolutely useless. A major problem that MIT has had in the past regarding software development efforts in the architectural school is that because all the projects have been small in scale, there is very little continuity from one year to the next as graduate students come and go. Quite often software that is started is never brought to the point of being a finished product, simply because to do so is much more than one graduate student can possibly do in the time he is there- unless he is a Ph. D. student. The next guy to come along doesn't want to spend his time wrapping up the last guy's project as he's under alot of pressure to experiment with new ideas which he can then use in his own thesis work. Therefore because it is not a team-programming environment like the real-world, many good ideas are never realized to their true potential. Most code that gets written is very mediocre in nature because it is never taken beyond the level of a "first cut", and program maintenance is a major problem. Knowing just how ominous this problem is, I knew that in terms of my own work, I had to find some way to deal with this situation. Otherwise a year after I left, my programs would suffer the same fate as all the others. This was perhaps the most convincing argument that I should stay away from tight data integration. It was more than I could personally deal with in one year in terms of coming up with a finished product, and the chance that another student would pick it up and continue on with it after I left was practically nil, due to the pressure he would be under to come up with his own thesis ideas. Since Project Athena was paying for my education, I needed to be sure I could hand them something that worked at the end of the year.

Due to all of the issues and constraints talked about so far, a reevaluation of my initial Project Athena proposal direction seemed necessary. It became clear to me that my first objective ought to be to write some useful code for MIT building technology courses that had what it would take to survive in an educational setting. Therefore I diverted my attention from the more luxurious, and now less important previous endeavors like those of data integration, to that of the question of how best to code the modules I would be working on. The question of data storage was finally solved when I began to think more carefully about this issue with regards to the runtime envi ronment my software would ultimately end up in. Project Athena sooner or later would be making a full committment to a UNIX flavored environment of some sort. Even if Athena used MS-DOS in the interim period, all MS-DOS software would eventually have to port to UNIX- or go by the wayside. No doubt for the sake of getting students prepared for UNIX, the MS-DOS environment would be made to emulate UNIX as much as possible in terms of directory structure and available utilities. Since text files are the de-facto standard way of dealing with data in UNIX (7), along with the system's wide variety of utilities- all or most of which are designed to work with text files, it made alot of sense to me to use text files as my primary means of data storage. This would provide an extra backup margin of flexibility beyond what I already knew through my previous experiences, I could do with text files myself. Given the proposition that the software environment would be in a perpetual state of change- due to new software packages coming and going such as new graphics packages and databases- all of which would be designed to interface with UNIX, it made sense to me to take advantage of this extra flexibility rather than trying to go against the UNIX grain. While this might be inefficient in terms of file size, it afforded a built-in level of continuity, assuring that code could be written in a "plug-in, plug-out" kind of fashion which is the UNIX standard (8). Moreover, since UNIX utilities and text editors could be used to view data, it saved me the problem of having to use intricate data decoding schemes within modules just for this purpose. All data would reside in one primary form, which was most compatible with the overall system environment.

I don't want to underestimate the importance of the implications I saw. Data consistency with other operating system utilities and applications programs meant that the software I wrote was likely to have a much longer life span, given the problems I have alluded to regarding software development within the architecture department in the past. Even though I was only going to develop some prototypical building technology software, data consistency in this flexible fashion, provided the likelihood that my software could become a foundation which could be built upon, as long as it was coded properly.

At this point, I felt that I had a tangible grip on my task of designing a building technology software system for architects, because I had finally crossed the intervening philosophical bridge, which gave me a focus on my direction. Since I now had a concrete conceptualization of my system, the remaining work simply involved filling in all the details which fulfilled all the conceptual requirements which I now had nailed down. In a sense, it all seemed very ironic to me since I didn't do much but to react to the larger circumstances I was faced with, but again, this is what designing in the real-world is all about. The direction the flagship was headed seemed contrary to my already determined need for flexibility- which of course meant that I required a large amount of autonomy, not a bunch of predetermined rules and regulations to follow. What seemed appropriate from the bridge of the flagship, did not look well from the waterline at all!, and seemed to me even to be antithetical to the spirit of UNIX and C. Consequently, I had come up with my own alternative- i.e.- the "cruise missile approach", which I found to be more logically consistent not only with the programming environment that I would be working in, but also with the overall goals of my project, which demanded that my software system have the same flexibility and tenacity as UNIX. Therefore, it only made sense that the UNIX/C environment should become somewhat of a model for my own system, particularly in view of the fact that UNIX has a substantial track record in higher level educational institutions and in research labs all across this country (9). Which I guess you could say, was my next hypothesis.

CHAPTER 4: RUNNING WITH ENDURANCE.

"Therefore, since we have so great a cloud of witnesses surrounding us, let us also lay aside every encumberance, and the sin which so easily entangles us, and let us run with endurance the race that is set before us."

Given all the constraints imposed on this project by the "flagship", including the fact that part of the project had to be completed by a "mid-year demo" to Project Athena, there was no time to loose. A week after the fall semester began, what I had been informed concerning the current status of Athena foretold the rest, and it was clear that I had to get moving right away. The first order of business was to pick two applications that would be appropriate choices for modules which along with the building system's command shell and it's needed utilities, would comprise the prototypical building systems software system.

As I mentioned earlier, we had a fair number of programs to choose from since we had been getting quite a few donations to the Designer's Software Exchange. However, most of them left alot to be desired because they had been written very crudely in BASIC, and were the bare skeleton of a good program- i.e. "toy programs" from the Apple II vintage. There were one or two BASIC programs which were of very high quality, however, since their source code was without comments and they were quite lengthy, they would have been a real bear to translate into C. This included MICROLITE (1), a daylighting analysis program written here at MIT, and TNODE (2), a rather sophisticated thermal network analysis program written at Georgia Tech. My goal was to find some useful programs which would not take months and months to translate, since I could not devote that much time to such a preliminary activity. Also, I wanted to find two applications that would be able to work together well and provide a comprehensive and meaningful software package to the designer, via the notion of "cascading software" as I discussed before.

The opportunity arose when we received a FORTRAN program in the mail which was the original version of CALPAS1 (3), the forerunner of the CALPAS3 program I had used at the University of Washington. The code was strictly "bare bones" with no frills- not even a user interface. The program had been written for a mainframe environment, and expected that an input file would be created by some other means. The fact that the program was "bare bones" meant that it could only handle a small building with a single thermal zone, such as a medium-sized house. However, I did not consider this to be a major drawback considering that my goal was to experiment with a prototypical sort of software package- i.e. a real "first-cut". My immediate objective was to learn how to design and code two modules so that they could work effectively together within their charted environment, and were maintainable. If this were done properly, unlike the past software written in the architecture school, once the system worked, these "first-cut" modules could be "unplugged" and replaced by better, more sophisticated versions later on when someone had the time to do it. For the purpose of training students in the fundamentals of building systems design, it was undesireable to have modules with a full allotment of bells and whistles anyhow, since these would only serve to confuse the primary learning issues involved. The fact that the code was without any frills also meant that it would be alot easier to translate into C. Although due to the complexity of the undocumented algorithym, it still looked like a horrendous task. Definitely a lesser evil than a real production-oriented program though.

CALPAS1 was to be the second of two modules- i.e.- the more sophisticated module which would provide a thermal analysis of a building during the later stages of design, just as we had used CALPAS3 before. In order to get a grip on a building's potential energy usage during the early stages of design, a much simpler and much faster algorithym was needed. And because such a simple algorithym ought to be easily coded in BASIC, by chance we just happened to have such a program sitting in the software exchange. SOLPAS was a Solar Load Ratio program which had been written in BASIC for the IBM-PC (4). The beauty of SOLPAS was that it required very little input, it ran very fast- i.e.- on the order of a few seconds, and it gave output concerning the general energy performance level of a building, given the particular types of passive solar systems used. In other words, it was great at "ball-parking" the energy performance of a scheme early on during the schematic design stage when the plan was still loose, and not much was known about the building besides it's basic form and system types. Since the method incorporated weather data for so many cities, it also gave the designer the option of looking to see if a proposed design would work as well in different locations. This was definitely an advantage to those who were designing tract rather than custom houses.

The reason SOLPAS was so well adapted for preliminary studies at the schematic design level, was that it was based on a correlation method, rather than a more involved "first principles" method, such as an hour by hour direct simulation. The Solar Load Ratio method was invented at Los Alamos National Laboratories by Balcomb and company (5), after years of studying actual physical test buildings where various types of passive systems could be monitored over an extended period of time. Extensive data was collected and via statistical data reduction techniques, correlated with climatic data based on some underlying assumptions as to the thermodynamic principles involved in the particular passive system type. Extensive cross-checking was done via "first principle" simulation methods running on large mainframe computers, to arrive at the proper correlation equation which when it's variables were replaced by the correlated values applicable for a particular system type, would work for all systems at the test location within a reasonable percentage of error.

These correlations were then further extrapolated to other locations with different average weather parameters so that the method now works with a total of 219 cities across the continent.

At the heart of the method is the LCR, or Load Collector Ratio, which is the net building load divided by the solar collector area. In other words, this ratio expresses the potential for solar heating- a small LCR tells you that you can expect a large amount of solar energy savings relative to an alternate scheme which has a large LCR (6). Of course the actual amount of solar savings will depend on the system type, and the climatic conditions this building will undergo. Once this information is fed into the process, the SSF or Solar Savings Fraction can be calculated, which expresses on a percentage basis how much energy this building will save relative to an identical "reference building", where the solar collector has been replaced by an energy neutral wall which neither gains nor looses heat. Multiplying the Solar Savings Fraction by the net heating load and the degree days for a specific location, gives the actual solar contribution for that system. On a house comprised of more than one system, this calculation can be area-weighted to derive the contribution of each system. Conversely, mutiplying the net heating load and the degree days of the locaion in question by (1 - SSF) will give the annual auxilliary energy required to match the annual heating load. While this method is basically simple and doesn't give any elaborate information such as a "first principles" method would concerning the actual comfort within a building- which is the most important issue, it does at a glance, give the designer a good indication as to how one scheme will stack up against another at a specific building location (7). Which is of course precisely the kind of information which is needed and highly valued during the schematic stages of design.

Later on, once the design process has narrowed down the choices to a few attractive alternatives, more detailed information is desired as one begins to try to optimize each scheme in conjunction with the architectural requirements, to try to pick the winning design. Here's where a program like CALPAS1 comes in. Based on an hour by hour, "first principles" simulation process, CALPAS1 can actually give an indication of the actual comfort conditions within a building for any specified time period during the year (8). The method utilized is called a "network analysis" whereby a mathematical relationship is assumed among the various primary building components that make up the building. Each primary building component becomes an element in a mathematical matrix, specified by a series of thermal properties. These thermal properties determine theoretical heat transfer relationships between all the various elements of the matrix. Given some input to this thermal system, a matrix solution method is used to indicate how the system has changed- i.e.- what effect this input has had on all the corresponding system elements, given the complexity of interactions that occur among them- due to the fact that the overall system has changed state. A dynamic analysis is thereby performed whereby for every incremental time step during the period of analysis- which is normally a year, the after state of the previous time step of the system becomes the starting state for the next time step. At the beginning of each new time step, more input data is entered which specifies the assumed external affects on the building system at that particular hour of the year. This includes factors such as outside temperature, solar insolation levels, wind speed and direction, etc.... Thus at the end of a year, the thermal conditions of the various building nodes is known- i.e. - their final temperatures, as well as the final tally of accumulated heat transfer among the various nodes. In addition if it is asked for in the beginning, intermediate results can be gotten at any point in time within the period of simulation because intermediate conditions can be tallied as well.

Although the method is based on averaged weather data for a specific location, over the long haul the method is claimed to be fairly accurate. Of course whether the actual numbers are in fact on the mark or not, the program's biggest asset is that it allows you to "fine-tune" the basic design you have chosen via the simpler and quicker SLR method. Once the design has proceeded to the point where the designer must know details such as the appropriate number of glazings or the appropriate thickness of a floor slab or trombe wall, a network simulation method such as the one employed by CALPAS1 can be used to explore these nitty gritty questions. The criteria for success can be determined by the detailed temperature and energy use information that CALPAS1 puts out. Temperatures are predicted for both the space and the various building masses on a 24 hour basis, and heating, venting, and cooling energy totals are available on an hourly, daily, monthly, and annual basis. If the building has an acceptable comfort level during the cooler winter months, yet does not overheat during the summer or "swing seasons", and saves more energy than other design variations, then a design can be considered for all practical purposes to be optimized- providing it can still be built within the design budget.

Of course there is a heavy price to pay for such an extensive analysis. The original version of CALPAS1 takes on the order of twenty to twenty-five minutes to run once all the input data has been entered, which is a laborious process in itself since over ninety pieces of information are required. In other words, CALPAS1 is not the kind of program to use early on in the design process. Even if it's input parameters were somehow known at this early stage, the program is far too slow to be used in an interactive fashion. And even if this were not the case, in my opinion a network analysis program such as CALPAS1 isn't an appropriate design tool at this stage because the input required is too complicated. Rumor has it that Berkeley Solar Group, an outfit that has written more intricate versions of CALPAS1 for the microcomputer environment (they wrote the CALPAS3 program I used at the

University of Washington), has reduced the runtime period to between ten and twenty minutes via data compression techniques, and the same system of defaults I have spoken about earlier. Again, I stand by my earlier conviction that if defaulted and compressed data saves you ten minutes of runtime, it certainly won't make up for the time you spend twiddling your thumbs when you get your output back! In effect, trying to adapt a complicated algorithym to a rather simple design phase just does not work very well. It is better to use this type of algorithym exclusively for what it is good for, which is the detailed analysis.

Something must be done to alleviate the long hours it would take to do design optimization using such a tool which takes on the order of ten to twenty-five minutes to run however. From my way of thinking, once a designer gets to the point where he should in fact use this more advanced tool, he probably knows enough about the architectural design constraints to narrow down his possible design optimization options. Thus he probably does not have a whole smorgasboard of options to choose from at this stage in the design process, but only a few. Here's where a batch-processing environment would come in handy. Rather than waiting around to try these options in a consecutive fashion, why not load up CALPAS1 with a few variations of the main theme and run them all at once? He need not waste his time waiting around like in the old TEANET days, as long as the code is designed correctly so as to accomodate this type of facility. In the time it takes him to go out to a lunch meeting, CALPAS1 could be sitting there cranking through two or three design development variations. An analysis of the output after lunch via some at-a quick-glance graphics, would undoubtedly give some more design inspiration for perhaps the final fine-tuning. Load up the program with these variations before going home at five, and when he returns at eight the next morning, a final design development candidate will likely be waiting on disk for his final approval!

As I had mentioned earlier, my battle plan was to code the algorithyms of these modules separately so that they could be used independently of each other, yet in such a way that allows both input and output data to be "cascaded" from the SOLPAS to the CALPAS1 module. In effect, once the results of the simple analysis were known, they would be used to set up the more complicated analysis, thereby alleviating the user of some of the hassle of dealing with it's more complicated input data. The reason this system can work well is that although the SLR method doesn't require alot of input, many detailed assumptions are implicit in the correlation numbers for the various passive system types (9). This detailed data is precisely the kind of material specific information which the more sophisticated methods such as the network analysis used in CALPAS1 requires as input. Thus once the system type is chosen- which is exactly what the SOLPAS module is primarily good for, a whole series of inputs for the CALPAS1 environment can be set automatically. These inputs are the most "nitty-gritty" of them all, consisting of various property coefficients that the designer would other wise have to look up in a reference book on heat transfer, such as ASHRAE's Handbook of Fundamentals. Other information which is not specific to the system type can be "cascaded" as well, such as building dimensions, infiltration rates, etc... although some of this information is not exactly in the form that CALPAS1 can use. As much as possible, this information would have to be converted to the proper format, in order to reduce to a minimum the amount of data that has to be reentered through CALPAS1's own user-interface environment. Separate utility programs would be the purpose of training In other words, CALPAS1 would have an option which allowed data to be received via a SOLPAS file. Since this facility would not be needed during stand-alone CALPAS1 runs, the code required to do this data transfer would not be included in the body of the CALPAS1 program. Another sleek cruise missile would be around to do that specific job.

The whole crux to my plan was to write code as a series of built-up, modular parts which could be used alone or used together, just like designing a modular building. I like to think of it in terms of a "Tinker-toy" analogy: Tinker-toy components come in two basic variations. Each component either functions as a small strut- i.e. something that can be used to extend something that has been previously made alittle far ther, or as a joint, something that is used to join together a series of struts into an object. Of course since these compo nents are very generic in nature, many variations are possible simply by rejoining the components in a new manner. The more components are designed to be reusable via their generic nature, the more likely they will still remain serviceable in the future as new requirements dictate the need for new kinds of objects which had not been originally planned for. In this regard, the smaller and more compact the components, the more flexible and therefore usable they become as generic building blocks. The components themselves, can be used to create fancier components at the next level which can then be used as building blocks for more specific purposes, just as one can construct a geodesic dome from a series of triangles which in turn are made of a series of struts joined together, etc....

This is in essence, how I went about developing the code for these modules and utilities. The end result in my mind, is very much a software system with extensive "plug-in, plug-out" capabilities, composed of reuseable software components. The components are reuseable primary because a primary means of data storage was chosen i.e.- text files for the reasons I have mentioned before, and also because I have adhered very strictly to structured programming principles and the K & R coding style of designing very short functions which are themselves composed of other very short functions. In effect, the "small is beautiful" UNIX/C-coding concept, which differs dramatically from the more traditional coding concepts.

This is exactly the sort of "building block software system" that I am arguing is so necessary for software development to continue in a growing fashion, in an educational environment which has been plagued with all the problems that I have previously alluded to. You will notice that I said "structured programming", not "top-down design". While "top-down design" had to be kept in mind for the overall goal, once the need for specific pieces of code was established via this principle, the code itself was almost always coded "from the bottom up", via the use of very small components, or aggregations thereof, many of which had been coded previously for another purpose. While this necessary process may seem like an inherent contradiction, all I can say is that it again goes to show that software design is very similar to the "wicked design problem" that architectural design is (10), and that any further simplification into a more linear process in my mind can only lead to poorly designed software- just as it does to poorly designed buildings!

To illustrate what I am saying, I will briefly discuss some of the aspects of the code, aside from the basic algorithyms involved which have already been discussed. At the highest level, the system is broken up into several large modules, plus any utility programs which are needed to transfer data between modules. If the SOLPAS and CALPAS1 programs were comparable in size (they aren't now, but with future improvements to SOLPAS they probably will be), these large modules would roughly coincide with an input, calculation, and an output module for each program, plus a library of ascii and time-related functions which all these modules share as a resource of low-level components. In addition, each of the separate utilities are themselves coded as one module (See Fig. 1). Depending on the situation, several modules and/or utilities are needed to execute a task. All calculation results are transferred to system storage as text files where the next module invoked looks for it's input data. Any intermediate results used soley within a module are written to storage as a solid block of binary data via the fact that program variables are kept in unifying C "structures", since this data need not be accessible to other modules. While file I/O is in general a very slow process, and one would be inclined to think that this way of modularizing the system code would detrimentally affect performance- this is not the case, primarily for two reasons. The first is that ramdisk is used as the home of all utilities and files, as much as this is feasible. Since a data move in memory is many orders of magnitude faster than a read or write operation from a floppy disk , the normal agonizing waiting period one must endure with disk I/O is for the most part alleviated, as all files are relatively small. Secondly, modules that need to be loaded into memory from ramdisk, are loaded when a file is read in. Thus this additional operation is effectively masked. From a psychological standpoint, the user shouldn't get uptight about this extra process, since he or she isn't even aware that it's happening.

On the intermediate coding level, these large modules are

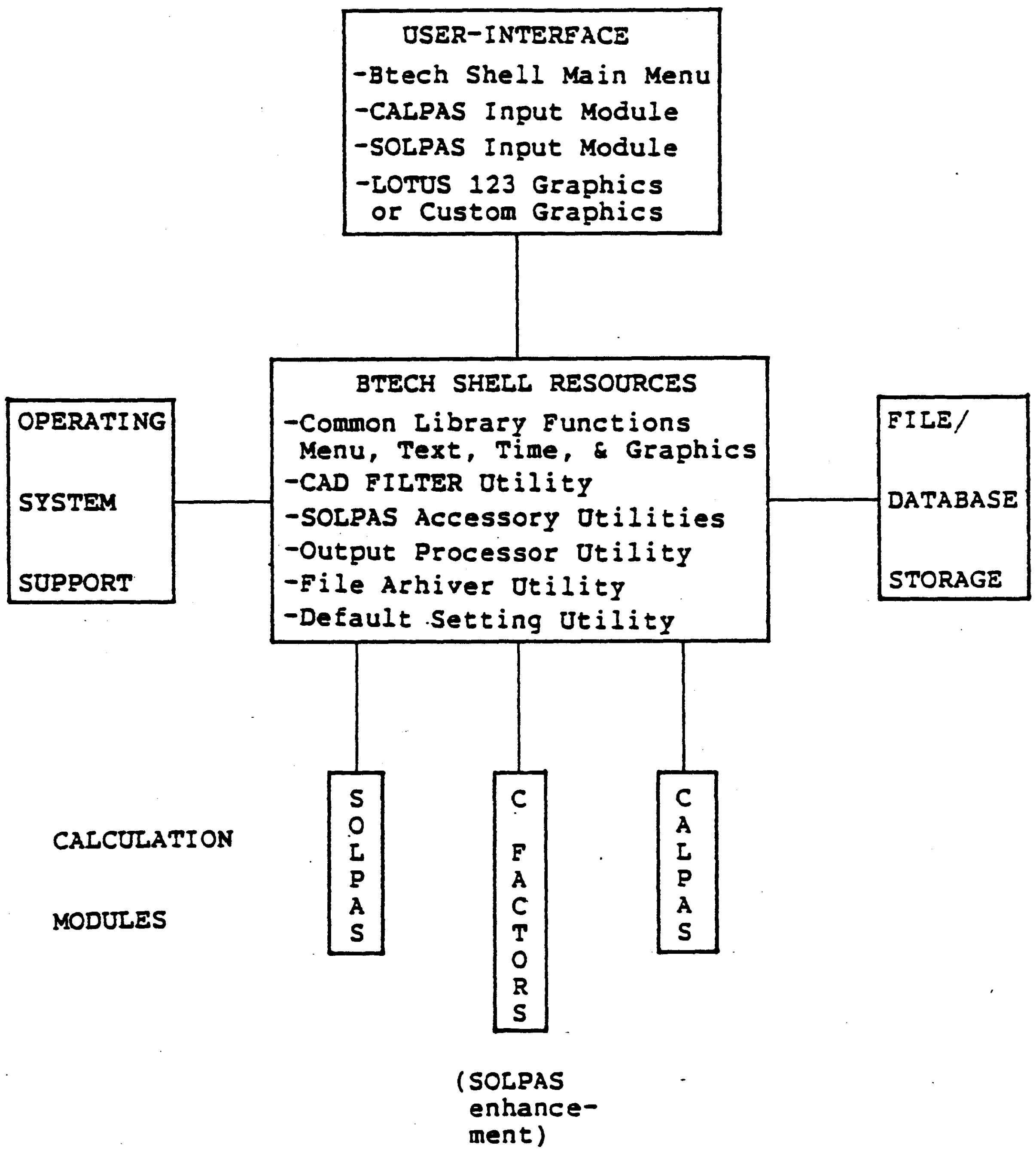


Fig. 1 A view of the software at the finished product stage.

broken down into smaller sub-modules using a unified means of handling data between these various functions- i.e.- structures, so that one function can easily be replaced with a new version- ala true K & R coding style. Because of a particular code organization technique which will be described below, a hierarchy of reuseable parts is generally available to all these modules via the libraries, including some built-up, higher level components which function as building blocks and can be further built up in various manners to do more complicated input and output operations.

Where things start to really happen however, is at the "component" level. The basic components of my system are in fact little routines which do primary operations on text strings. This worked out to great advantage because not only was the majority of stored data in textual form, so was the data that was required to be input via a user-interface. For the purpose of getting data into a module, I needed a series of building blocks which had the ability to read certain kinds of data. Some of this was straight characters or text, while the majority of it was text in the form of either integer or real numbers. For the purposes of flexibility and code maintenance as I mentioned earlier, a primary goal I had was to break the BASIC habit of intermixing screen I/O calls with the underlying algorithym, as this is primarily what makes these programs so difficult to translate and transport. My goal was to develop a series of component functions which made all this I/O, no matter how complicated it was- transparent to the program. This had the added advantage of allowing the incorporation of a better user-interface at a later date after the whole system had been designed. In other words, it allowed for a "second-cut". Consequently, after I chose the style of user-interface I thought was most appropriate for the time being, I went about developing component functions which could read data into the particular format required. While these components are quite advanced in terms of fancy cursor footwork, all of this lower level stuff is completely transparent. All the module itself sees, is an assignment statement which assigns the particular piece of data received to the proper variable. Thus the segment of code that enters a real number, doing all kinds of cursor footwork and error checking in the process is simply a statement of the form:

```
value = getreal();
```

where the function getreal() resides off in a separate source code library of text related functions. By extention, a whole input screen who's task it is to get all input relating to a specific aspect of a building, such as the south wall for instance, consists of a screen title, plus a series of simple assignment statements- and that's it! Nothing else to cover up the underlying algorithym. Thus the whole input section of a module is almost solely composed of a series of these simple screen functions which themselves are just a series of assignment statements- i.e.- the whole thing is just a series of building blocks, themselves composed of smaller building blocks.

Since user-interfaces are normally fairly complex, the reader might wonder about such things as prompts, defaults, and error messages. All of these are separate building blocks which are called into other building blocks as parameters. For instance, all SOLPAS and CALPAS1 input prompts are aggregated together in separate building blocks which are in essence arrays of text strings. All defaults of which there are three kinds in all- a lower limit, a current default, and an upper limit, are aggregated together as arrays of structures composed of these defaults. All program variables are arranged in a structure of structures as well- of which each substructure pertains to a particular screen. Not only does this allow the input data to be sent somewhere as a complete "packet", it also sets up a facility whereby a complicated input mechanism can be handled very simply via an assignment statement. This is because a one to one correspondence has been established between these types of building blocks. In essence, the components of building blocks of a particular type are all numbered, and all building block components with the same number relate together in some predetermined fashion. It is really a process of code integration. For example, the more exact call required to pull in a data item would be something as follows:

```
   value = getreal(mesg,def->low,def->high,def->cur,precs);
```

where each variable is in fact an array element, which currently is set to the same indice level (I purposely left off the indices so I could get the assignment statement all on one line). Multiply this one statement by six, add a title at the top, and you have yourself a complete screen function which reads in six pieces of data (See Figs. 2-3). A function where the user-interface itself is transparent, and in fact replacable at a future date with a different sort of user-interface, as long as the function calls involved appear to the module in the same fashion.

Written in this manner, the algorithyms of SOLPAS and CALPAS1 are completely decoupled from the input section, and in fact in CALPAS1 (SOLPAS was not big enough to bother with), reside in entirely different source files (See Fig. 4). In a similar building block fashion, after the input sections do their job, one line does it all:

```
calculate();
```

whence you are finally ready for the output section which is set up in a similar fashion to the input section. (Calculate() however, is composed of many building blocks itself in terms of logical subdivisions of the underlying calculation algorithym.)

The output section is built up of various components which in general, couldn't care less where the output was

```
southwall()
{
    char string[20];

    while (TRUE) {
        scr_clr();
        scr_rowcol(0,31);
        puts("SOUTH WALL MODULE\n\n");
        sth[0] = getintgr(mesg[8],def[0]->low,def[0]->high,
                              def[0]->cur);
        sth[1] = getintgr(mesg[9],def[1]->low,def[1]->high,
                              def[1]->cur);
        sth[2] = getintgr(mesg[10],def[2]->low,def[2]->high,
                              def[2]->cur);
        sth[3] = sth[0] * sth[1];
        itoa((long)sth[3],string);
        printf(mesg[11],string);
        putchar('\n');
        if (getintgr(mesg[69],(long) 0,(long) 1,(long) 'y')) {
            southzones();
            break;
        }
    }
}
```

Fig. 2 SOLPAS code which asks for south wall input, shows function calls which detach the user-interface from the underlying algorithym.

```
long getintgr(prompt,low,high,def)
char *prompt;
long low,high,def;
{
    char string[20],buf[20],strlow[12],strhigh[12],rng[40],c;
    static char backup[10] = "range:  ",rng1[5] = " to ",
                rng2[7] = "y or n";
    static char yes[2][4] = {"yes","y"};
    static char no[2][3] = {"no","n"};
    int i,row,col,beg=0,end,save,invalid=TRUE,num=TRUE;
    long var,fatol();

    if (def == 'y' || def == 'n') {
        num = FALSE;
        strcpy(rng,backup);
        strcat(rng,rng2);
    }
    else {
        strcpy(rng,backup);
        itoa(low,strlow);
        itoa(high,strhigh);
        strcat(rng,strlow);
        strcat(rng,rng1);
        strcat(rng,strhigh);
    }
    string[0] = '\0';
    row = scr_srow();
    col = scr_scol();
    while (invalid) {
        if (!string[0]) {
            if (num)
                itoa(def,string);
            else {
                string[0] = ' ';
                string[1] = (def == 'y') ? ('y') : ('n');
                string[2] = '\0';
            }
        }
        else {
            scr_rowcol(row,beg);
            puts("     ");
        }
        scr_rowcol(row,col);
        printf(prompt,string);
        end = scr_scol();
        beg = end - strlen(string) + ((def < 0) ? (0) : (1));
        scr_rowcol(row,beg);
        if (get(buf,'i') == XERR)    return(def);
        if (isalpha(buf[0])) {
            scr_rowcol(row,beg+strlen(buf));
            puts("     ");
            if (num) {          (See caption on next page)
                scr_rowcol(row,RNGLOC);
                scr_aputs(rng,REV);
                continue;
            }
            for (i=0; buf[i]; i++)
                buf[i] = tolower(buf[i]);
            buf[i] = '\0';
            for (i=0; i<2; i++) {
                if (strcmp(buf,yes[i]) == 0) {
                    scr_rowcol(row,RNGLOC);
                    scr_aputs("                              ",NRML);
                    putchar('\n');
                    return(1);
                }
                else if (strcmp(buf,no[i]) == 0) {
                    scr_rowcol(row,RNGLOC);
                    scr_aputs("                              ",NRML);
                    putchar('\n');
                    return(0);
                }
            }
            scr_rowcol(row,RNGLOC);
            scr_aputs(rng,REV);
            continue;
        }
        if (buf[0]) {
            scr_rowcol(row,beg+strlen(buf));
            puts("     ");
            if (!num) {
                scr_rowcol(row,RNGLOC);
                scr_aputs(rng,REV);
                continue;
            }
            var = fatol(buf);
            if (var < low || var > high) {
                scr_rowcol(row,RNGLOC);
                scr_aputs(rng,REV);
            }
            else {
                scr_rowcol(row,RNGLOC);
                scr_aputs("                              ",NRML);
                putchar('\n');
                return(var);
            }
        }
        else {
            scr_rowcol(row,RNGLOC);
            scr_aputs("                                    ",NRML);
            putchar('\n');
            return( (!num) ? ((def == 'n')? (0):(1)) :(def));
        }
    }
}
```

Fig. 3 This function which handles screen interfacingfor integers resides in a separate source file,hiding tricky code from SOLPAS's algorithym.

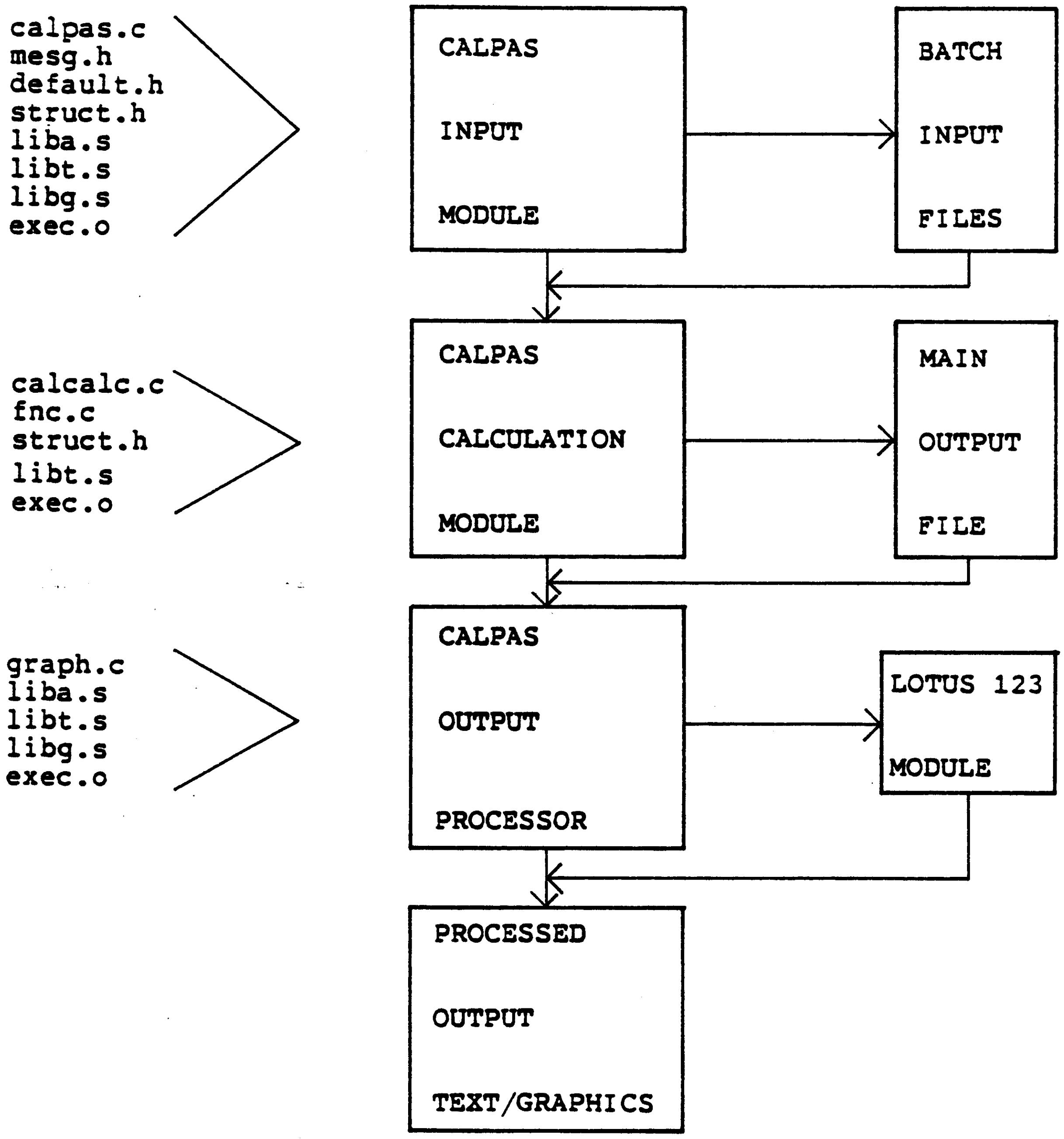


Fig. 4 Diagram of the CALPAS module showing the different reusable building blocks which are used.

going, whether it be to the screen, the printer, or a file. The simpler components simply write a line of text out to an output device. More complicated building blocks of such components provide fairly extensive formatting facilities, for example the ability to reformat a screenful of output data in a tabular form, grabbing data from a larger, more complete ascii data file. These processes are table driven to allow for future new accomodations and updates. This is precisely the technique that was used in the CALPAS1 output processor, which also serves a dual function as the graphics preprocessor to LOTUS 123 (the only canned utility used), or to any other outside spreadsheet or database. The philosophy behind the development of this all-round utility was as follows:

If the reader will remember, CALPAS1 puts out an enormous amount of varied output, including several ranges of temperatures, and many heating, venting, and cooling energy tallies for various time periods during the simulation year. From one run to the next, the user could obviously choose amonst all the various options as to what he or she wanted to see, graph them via LOTUS 123, and/or archive them for later reference. Of course the BASIC way to handle something like this, would be to put out a whole bunch of little files in binary, each pertaining to one of these options. These little files can of course be aggregated together via a header into one big amorphous file, yet the outcome is the same- one huge mess where output files can vary considerably.

It was also very conceivable that at some time in the future, new forms of output from either SOLPAS or CALPAS1 would be desirable. For instance once the system was up and running, many enhancements could be "plugged into" SOLPAS including the ability to handle something called "conservation factors", and some means of performing a quick check on cooling needs. Thus additional data would need to be displayed and written to storage in a manner which is transparent to the existing module, so that no code needs to be rewritten. This additional output would not always be used during every simulation, and would increase the amount of output to a point where it would take several screens to display it all. In the case of CALPAS1, if all the necessary screens had all been "hardwired" into the code, the code would have been encumbered with about ten separate routines which formatted pieces of output and spewed them onto the CRT. At the same time, after each of these screen images had been sent off to storage, some means of recalling them at a later date and redisplaying them against other data archived from other runs was necessary. This brings up the necessary aspect of file merging- providing some means to aggregate chunks of archived data from different simulation runs so that the results can be viewed together. Thus some means of perusing data files at large in order to pick out the various segments of archived data which one would want to view together was needed as well.

If this data were stored as ascii characters in the same format in which it originally appeared on the screen, a simple word processor or screen editor could be used for this file perusal task. In addition because the data is in textual format, under UNIX, a simple CAT command would concatenate two or more text files together at the operating system level providing a painless file merging service. Also available at the UNIX system level are numerous search, find, and count utilities which could be used to help find data files which contained specific text data (11). And last but not least, the fact that data stored in field-formatted, text fashion is easily read into any spreadsheet or database program (12), provided the clincher that an ascii data format was the way to go, just as I had anticipated earlier.

So the problem became, how do you store all this information as text, and yet have the ability to pick out at any given moment exactly what the user wants to see ? The answer is of course some kind of utility which can preprocess an ascii file before it's data is used. Thus the CALPAS1 output processor was written as a separate module which like a general utility could be used for a broader scope of purposes. The same reusable components that make up this utility were also used in the utility which processed CAD drawing program output files for attributes and sent them to SOLPAS, since that process involved formatted text data as well.

Central to the whole concept behind these utilities is the notion of buffered file I/O. Because of all the variations possible with output options, it was never possible to know in advance how long a given file would be. Thus the building blocks of these utilities would have to be able to process chunks of ascii data, and yet be able to go back and find any remaining file chunks and begin processing them exactly where it left off. Thus buffering was needed. At the same time, it was not possible to know in advance how much memory was available to be used as a buffer, since the various modules and utilities running within the system needed various amounts of memory in order to run. Sometimes several of these processes would have to be loaded into memory at the same time, such as when LOTUS 123 is called up by SOLPAS for the purpose of graphing output data. Thus depending upon the applications modules these building blocks were to be used with, the size of the buffer had to be redefinable. This feature was implemented and later proved to be a saving grace. In processing CALPAS1 output data, the most logical buffer size was 8192 which is a multiple of 512 (a performance advantage) that is big enough to be able to handle the biggest file CALPAS1 was likely to put out. However, later when the link to LOTUS 123 was established, this size proved to be too large, as the operating system did not have enough memory left to be able to load 123.EXE behind the output processor. Changing one line in the output processor source code which reduced all references in the code to a new buffer size of 512 bytes easily adapted the utility so that it would work in this situation. And no performance disadvantage was noticed at all!

Since the first record in a line of CALPAS1 output data always refers to a time within the simulation year, whether this be in reference to an hour, day, month, or year, the ouput processor runs off a pattern-matching component which searches the buffer for all the output pertaining to the time period in question. Once this data is located, various other components have the ability to pick off patterns of ascii data, converting them into appropriate numbers based on a table-driven process which records which pattern in the text is for what purpose. In other words, if you are looking for monthly cooling data, the processor knows that once it finds monthly data, cooling data happens to be in the 8th column over from the left, etc.... (See figs. 5-7)

Due to the fact that this table-driven pattern-matching method has inherent flexibility, it is a rather trivial matter to compare output which resides in different run files. Just CAT the two files together first- then run the result on the output processor module. As long as the proper keys were in both files before they were concatenated together, the proper data can be drawn out and compared. While this process is basically a slow one utilizing sequential searches, since the files involved are always small- i.e. less than 10k in length, the time involved is trivial, especially if the files reside in ramdisk. One can hardly tell if the split-second taken is a consequence of the searching procedure, or just due to the fact that screen I/O is a very inefficient process. SOLPAS as it stands now only puts out three columns of output data for each month of the year, so at the time I coded this module, I did not bother incorporating in an output processor such as

```
***********************************************
HOURLY SUMMARY:

TIM    TZ      HTD    HVTS      SHG     TA   TAIR  TS21   TS15    TS3

385  39.0      0.0     0.0      0.0   70.0   70.0  71.9   65.0   65.0
386  38.0      0.0     0.0      0.0   69.3   69.3  71.5   65.0   65.0
387  37.0      0.0     0.0      0.0   68.6   68.6  70.9   65.0   65.0
388  35.0      0.0     0.0      0.0   67.8   67.8  70.5   65.0   65.0
389  35.0      0.0     0.0      0.0   67.2   67.2  69.9   65.0   65.0
390  35.0      0.0     0.0      0.0   66.8   66.8  69.4   65.0   65.0
391  35.0      0.0     0.0      0.0   66.5   66.5  69.0   65.0   65.0
392  37.0      7.3    15.2    196.1   66.9   66.9  68.7   65.0   65.0
393  36.0     51.6    81.2    878.5   67.8   67.8  68.9   65.0   65.0
394  40.0    100.9   147.8   1257.8   69.3   69.3  69.4   65.0   65.0
395  43.0    135.2   191.4   1252.5   71.1   71.1  70.3   65.0   65.0
396  47.0    151.2   211.8   1027.1   72.4   72.4  71.2   65.0   65.0
397  47.0    149.1   211.4   1721.2   73.7   73.7  72.2   65.0   65.0
398  50.0    129.9   194.6   2254.7   74.7   74.7  73.1   65.0   65.0
399  52.0     94.4   153.9   2430.8   75.3   75.3  73.8   65.0   65.0
400  52.0     53.2   100.9   2127.8   75.4   75.4  74.2   65.0   65.0
401  52.0     16.0     9.6    262.1   75.0   75.0  74.1   65.0   65.0
402  51.0      0.0     0.0      0.0   74.6   74.6  74.0   65.0   65.0
403  50.0      0.0     0.0      0.0   74.3   74.3  73.7   65.0   65.0
404  49.0      0.0     0.0      0.0   73.7   73.7  73.5   65.0   65.0
405  48.0      0.0     0.0      0.0   73.2   73.2  73.3   65.0   65.0
406  48.0      0.0     0.0      0.0   72.9   72.9  73.0   65.0   65.0
407  44.0      0.0     0.0      0.0   72.2   72.2  72.7   65.0   65.0
408  44.0      0.0     0.0      0.0   71.3   71.3  72.4   65.0   65.0
```

Fig. 5 The portion of CALPAS's output file showing hourly temperature data which the Output Processor can find, reformat, display and graph. The processor skips every other line of data in the file to retrieve hourly temperatures at two hour intervals.

CALPAS HOURLY OUTPUT
SPACE & STORAGE TEMPERATURES

HOUR OF DAY	AIR TEMPC	WALL TEMP	FLOOR TEMP	QUICKWALL TEMP
2am	69.3	71.5	65.0	65.0
4am	67.8	70.5	65.0	65.0
6am	66.8	69.4	65.0	65.0
8am	66.9	68.7	65.0	65.0
10am	69.3	69.4	65.0	65.0
12am	72.4	71.2	65.0	65.0
2pm	74.7	73.1	65.0	65.0
4pm	75.4	74.2	65.0	65.0
6pm	74.6	74.0	65.0	65.0
8pm	73.7	73.5	65.0	65.0
10pm	72.9	73.0	65.0	65.0
12pm	71.3	72.4	65.0	65.0

Fig. 6 This is the table of output that is produced from the data in Fig. 5 by the Output Processor.

```
stortemps()
{
    int i,lncnt=0,match,r;

    scr_clr();
    scr_rowcol(0,0);
    for (i=0; i<2; i++) {
        fputs("\n                                   CALPAS HOURLY
             OUTPUT\n",fd[i]);
        fputs("                                SPACE & STORAGE
             TEMPERATURES\n\n",fd[i]);
        fputs("         HOUR OF          AIR           WALL
             FLOOR        QUICKWALL\n",fd[i]);
        fputs("             DAY          TEMPC         TEMP
             TEMP     TEMP\n\n",fd[i]);
    }
    zerobuf(buf,0,BSIZE);
    charptr = 0;
    if (fread(buf,1,BSIZE,fd[3]) == 0) {
        printf("\nError reading input file: %s\n\n",file[0]);
        return(0);
    }
    match = fhr[0];
    while (!(r = locate("TIM ",match,fd[3])))
        ;
    if (r == ERR)    return(0);
    charptr = save;
    id[0] = 0;
    id[1] = 7;
    id[2] = 8;
    id[3] = 9;
    id[4] = 10;
    pad[0] = 4;
    pad[1] = 2;
    pad[2] = 2;
    pad[3] = 2;
    pad[4] = 2;
    while (TRUE) {
        if (!getline(fd[3]))         break;
        if (getline(fd[3]) && getrtvals(id,tag,5)) {
            time(atoi(tag[0]),frec);
            putvals(tag,frec,pad,1,5,0,fd[1]);
            ++lncnt;
        }
        else    break;
    }
    fputc('\n',fd[1]);
    fputc(EOFC,fd[1]);
}
```

Fig. 7   Main C function used to create the above table of storage temperatures in Fig. 6 from the CALPAS output file data shown in Fig. 5.

the one CALPAS1 uses. However, when the new features are added down the road that I have previously spoken of, the fact that these building blocks already exist, will prove to be a saving grace again.

Indeed, this "tinker-toy" endeavor has already proved to be a saving grace in terms of extending the useful life of my code, just as I hypothesized. For the "midyear demo", it was deemed prudent to have an application running that could pull in information that had been previously stored via a CAD drawing package. Therefore I wrote the FILTER utility which is the name of the module which grabs data from a drawing program attribute file and sends it off to SOLPAS. On the drawing program end, this process is very inefficient, and since our original experimentation with it, the flagship crew has decided that it would be much better for the program to put out a simple indexing key to an external database where this bulky data could reside. This frees up the drawing program workspace considerably since all this unnecessary garbage is off somewhere else, giving the program full rein to do it's thing, which is drawing. Luckily, because of the building blocks involved, the components of FILTER couldn't care less where the data resides as long as it's in a text formatted form, and it should be a rather trivial matter to amend this utility so that it grabs it's data from a database package instead, which has this formatting facility (See Figs. 8-10). Matter of fact, as long as the database software has an ability to make it's record format known to outside

```
char attrtag[10][20] = { "name","special","length","width",
                         "rvalue","ohx","ohy","glzarea",
                         "numglz",""};

char key[7][20] = { "south","east","west",
                    "north","roof","floor","" };

char numdef[20] = "-999";

char exttemp[20] = ".txt";

char extattr[20] = ".txt";

char extstruct[20] = ".str";

char extdbase[20] = ".atr";
```

Fig. 8 All program variables which refer to tag names and indexing keys which are specific to a par-CAD program or database, are in a separate source file from the FILTER main program to facilitate easy maintenance.

```
extern char attrtag[][TAGSIZE];
extern char key[][TAGSIZE];
extern char numdef[];
extern char exttemp[];
extern char extattr[];
extern char extstruct[];
extern char extdbase[];

char tag[MAXTAG][TAGSIZE] = { "bl:level","bl:name","bl:x",
                              "bl:y","bl:layer","bl:orient",
                              "bl:xscale","bl:yscale" };

struct tmp { char nname[TAGSIZE];
               char aattrr[TAGSIZE];
               char ffmt;
               int field;
};

struct tmp template[MAXTAG];

struct data { union {
                        char sval[TAGSIZE];
                        real nval;
                        } uval;
                        char utype;
};

struct data _blevel;
struct data _blname;
struct data _blx;
struct data _bly;
struct data _bllayer;
struct data _blorient;
struct data _blxscale;
struct data _blyscale;
struct data _attribute1;
struct data _attribute2;
struct data _attribute3;
struct data _attribute4;
struct data _attribute5;
struct data _attribute6;
struct data _attribute7;
struct data _attribute8;
struct data _attribute9;

struct data *dataptr;
```

(See caption on next page)

```
struct obj { struct data _blevel;
                   struct data _blname;
                   struct data _blx;
                   struct data _bly;
                   struct data _bllayer;
                   struct data _blorient;
                   struct data _blxscale;
                   struct data _blyscale;
                   struct data _attribute1;
                   struct data _attribute2;
                   struct data _attribute3;
                   struct data _attribute4;
                   struct data _attribute5;
                   struct data _attribute6;
                   struct data _attribute7;
                   struct data _attribute8;
                   struct data _attribute9;
};

struct obj object1;
struct obj object2;
struct obj object3;
struct obj object4;
struct obj object5;
struct obj object6;

struct obj *objptr;

struct group { struct obj object1;
                     struct obj object2;
                     struct obj object3;
                     struct obj object4;
                     struct obj object5;
                     struct obj object6;
};

struct group all;
```

Fig. 9 "Universal data structure" used by the FILTER utility to store number or text attributes from a CAD drawing or database. This data structure is hierarchical and independent of attribute tag names, since any element can be accessed by pointer arithmetic. It currently holds 17 attributes for 6 surfaces of a building, but can be easily extended.

```
senddata()
{
    FILE fd,fopen();
    int i,j;
    char string[20],file[20];

    objptr = &all;
    dataptr = objptr;
    strcpy(file,attrfile);
    strcat(file,extstruct);
    if ((fd = fopen(file,"w")) == 0) {
        printf("\nCan't write output file:  %s\n",file);
        puts("....  program aborting!\n\n");
        return(0);
    }
    fwrite(all,1,sizeof(all),fd);
    fclose(fd);
    strcat(attrfile,extdbase);
    if ((fd = fopen(attrfile,"w")) == 0) {
        printf("\nCan't open output file:  %s\n",attrfile);
        puts("....  program aborting!\n\n");
        return(0);
    }
    for (i=0; i<numkey; i++,objptr++) {
        dataptr = objptr;
        for (j=0; j<numtag; j++,dataptr++) {
            if (id[j] != ERR) {
                if (dataptr->utype == 'n') {
                    ftoa(dataptr->uval.nval,(double)
                         template[id[j]]->field,2.,string);
                    fprintf(fd,"%s",string);
                }
                else if (dataptr->utype == 's') {
                    fprintf(fd,"%s",dataptr->uval.sval);
                }
            }
        }
        fputc('\n',fd);
    }
    fclose(fd);
    return(1);
}
```

Fig. 10 A simple function that stores the attribute data on disk using pointer arithmetic. Two versions are stored, one in binary, and one as field-formatted text for screen display, spreadsheet/graphics input, or database storage.

software, no adaptation to FILTER will be necessary at all.

One last word on the subject is that these same building blocks have been used to improve the performance of CALPAS1 substantially over the original FORTRAN version, in terms of the reading of it's weather file- which is a mammoth 270k! The original version used a FORTRAN routine which actually had to read the next record in the file each time it looped through it's calculation routine- a total of 8,854 iterations! Consequently, if this file was stored on floppy disk, the drive was running continously. Being by far the slowest operation, the program forever needed to read the drive again to retrieve the next hour's weather data. Talk about an inefficient process! With my ascii building blocks, I replaced this piece of code with a routine which not only buffered the weather data in 23k chunks (i.e.- a whole months worth of data was read into memory at a time where it was subsequently processed), it also allowed the weather data to reside in a "free format" where each element of weather data no longer had to occupy an exact byte location within a data record. This provided a considerable amount of necessary freedom for the poor guy who ends up porting weather files from other applications which can be used by CALPAS1. As a result, instead of having to read the disk 8,854 times during a simulation, my new version only has to read the disk 13 times! Since I never had the patience to run a comparison between these two versions off of floppy or hard disk (I've always used ramdisk where the difference is not that noticable), I'm not sure what the performance advantage is, but it must be considerable.

A case like this by the way, is where I think performance should be given close attention. The extra time taken to load up an extra utility module in ramdisk to massage some data is negli gible. However, this is not the case when an operation occurs within a huge calculation loop which gets executed 8,854 times. The moral? -- put your coding effort where it really counts!

For the time being, rather than adding more system features, I am concentrating my efforts on porting my code to a Lattice C environment, so that our flagship commander will be happy after I am no longer around MIT. Unfortunately, Athena did not discover in time before it made it's selection, that DeSmet had a much cheaper and tighter C software development package, so I am forced to port over to Lattice. But at least I was able to get a significant amount of code written this year, in a manner which hopefully has laid a good foundation for future building technology software development efforts at MIT. This is of course due to the strict adherence to structured programming principles, and the coding of vast libraries of reusable "building blocks" which can be utilized by various software modules. As I mentioned above, I already have an inkling that this particular coding technique- indeed will lay a foundation by which other software developers can build on, providing that they go along with my "cruise missile" approach, and use text files as a compatible footing on which building blocks can be laid. My success in writing integrated code, laying building blocks upon other building blocks, and my ability to reuse these components across various modules and utilities seems to point to the fact that the system as I have developed it does in fact have UNIX-like tenacity, due to it's inherent "plug-in, plug-out" capabilities. The fact that the CALPAS1 ouput processor is such a flexible and generic tool which can easily accomodate future needs, even some which may as yet be unanticipated, in my mind adds a significant amount of hope that this type of software system can survive the hazards of an educational environment, just as UNIX has been able to. The fact that UNIX is the local operating system here at MIT adds even more hope, due to the compatibility of these systems on two levels- i.e.- at the conceptual level in terms of coding philosophy, and on an implementation level, due to the fact that they have the same means of primary data storage.

Maybe there can now be more continuity, and some real growth here in the future- as now I can say to the next MIT graduate student who comes along: "Here is your foundation to build on, if you will continue the work I have started, your own work will be alot easier, since I have already created all these building blocks. Now it is all up to you!"

CHAPTER 5: THE RESULTS

"All discipline for the moment seems not to be joyful, but sorrowful; yet those who have been trained by it, afterwards it yields the peaceful fruit of righteousness."

At this point in time, this comprehensive passive solar software environment- which is the subject of this thesis, lacks a few modules to be complete. This is besides all the fancy enhancements that I've said will probably be added down the road to SOLPAS. The file archive facility which in some fashion will keep track of a user's various run files has not been coded. It is the intention that this facility will enable one to keep track of his or her files as one uses this software day by day on a design project. When one stores a file to disk, the opportunity will arise to store this file in the current working directory, or to store it more permanently in a file archive where one has the ability to associate the name of the file with a string of text, which at a glance will help one to identify it. An eight charater filename simply isn't long enough to do the job properly, as it doesn't take very many files for this method to get out of hand, as we all know. The ability to replace a filename such as "SG200R11", with a phrase such as "3rd variation w/ 200sf south glazing and R11 insul." should relieve a significant amount of user frustration. It is also anticipated that this facility will have the power to associate a run file with it's various counterparts such as the associated weather and input files which were significant in it's creation. Whether this facility will take advantage of existing canned software such as a spreadsheet or database, or be written entirely in C, I do not know at this point since I have not really though about all the implications. Either way, the building blocks are lying around somewhere to get the job done when I finally get around to it.

The last piece of code is the BTECH shell which is the "housing" where all these modules and utilities will exist. It will most likely be just a simple menu interface shell which will give the user the choices of running SOLPAS or CALPAS1, letting him peruse his database of existing files via the FILE ARCHIVER or his own favorite screen editor, or letting him display and graph various output files via the OUTPUT PROCESSOR- whether or not he actually intends to run any new simulations or not. Since the current CALPAS1 output processor already contains all the building blocks that are necessary for such a LOTUS-type menu front-end, coding it should not be a difficult task once all the other modules are in their completed form.

Other features will be "plugged-in" to the system later as time and motivation levels permit. Harvey already has a student working on one enhancement to SOLPAS- i.e.- incorporating SLR "conservation factors" into the scheme, which will enable a designer to balance passive solar stategies against energy conservation strategies in the design optimization process. Other enhancements to SOLPAS are planned, such as adding Phil Nile's cooling algorithym as a quick schematic design check against an overheating problem. As far as CALPAS1 is concerned, the original version had the major drawback that only one sequence of detailed output data could be asked for in the input process. In other words, if one wanted to see hourly data for january 15th and also for june 15th, one's only choice was to specify all hourly data between these two dates- i.e. there was no way to specify more than one range of output data. To date, all of the low-level building blocks needed to alleviate this problem have been coded, but I have not gone back and incorporated them into CALPAS1's user-interface yet. But in the future, if you say that you want only output for january 15th, june 15th, and for the days of october 3 thru october 6- so be it, that's exactly what you'll get!

While my attempt to design and code a comprehensive software package to aid in passive solar design has alot of loose ends right now, the main modules which comprise the primary working elements of this software system have been fully coded and debugged at this stage. As the system currently exists, it is possible with alittle more effort than will be required in the end, to go through a complete passive solar design problem using the output from SOLPAS to generate appropriate input for a CALPAS1 run. The results of either method can be displayed graphically as well as tabularly, using LOTUS 123 as a graphing utility. Once output is graphed using this means, the user is free to continue on in a cyclical fashion revising and rerunning his proposed design in an attempt to optimize the performance of the passive systems involved. As I mentioned above, the only major differences between the current product and the finished product at this point, is the lack of the FILE ARCHIVER, and a user-friendly BTECH shell main menu which depicts all the system options. The utility which will automatically set CALPAS1 input defaults based on the outcome of a SOLPAS run has not been implemented either, but this operation can easily be done by hand through the normal CALPAS1 user-interface. Cascading the data is entirely possible at this stage, however, it may take a few extra minutes during the CALPAS1 input process. Since CALPAS1 need not run directly behind SOLPAS, it is anticipated that this cascading utility once it is developed, will take advantage of the FILE ARCHIVER's ability to pull up run files which have been run at a previous time, which is the reason I am holding off on the coding of this final utility.

I think it is possible however at this stage to get a pretty good glimpse of how the system will perform, and how it may be an improvement over past building technology design tools. The first thing to be said, is that the system is indeed coherent, both from the standpoint of the user, and from those who will end up having to maintain the software in the future. In the last chapter I alluded to coherence with respect to software maintenance, since everything within the system has been coded with the same reusable "building blocks". This chapter describes the system in respect to it's use, and how the system tries to maintain a "logical coherence" from the standpoint of the user. Performance in terms of ease and efficiency of use is also discussed.

An important consideration with respect to "logical coherence", is how the individual modules within the system look with respect to each other. I went through an extensive amount of effort to make sure that the SOLPAS and CALPAS1 modules appear almost as twins in terms of how the user is required to interact with them. In other words, if the user is already familiar with one of the modules, he or she should have absolutely no trouble using the other. This in itself, is a marked improvement over building technology software that has been previously designed as I mentioned in the introductory chapters of this thesis. What I mean when I use the word "twins", is that in every aspect, one module emulates the other in respect to how one interacts with the user-interface- when program options are asked, when file I/O takes place, how the graphing and parametric analysis features are invoked, etc..... If it were not for the fact that each module when run comes up displaying it's own identification banner, the uninitiated user would have a very hard time telling the two modules apart! In other words, the fact that the these modules are consistent on the inside at the coding level, has given them a consistency on the outside as well- which was one of my original hypotheses.

The particular style of user-interface that I chose to implement is very consistent with UNIX operating system style as well. In other words, it is alittle on the terse side, but this was entirely intentional for several reasons. I might add here that a few people have commented on the fact that this interface is indeed terse, and wondered why I did not implement something which was a little more "user-friendly". My general comment in return, is that I think the system is user-friendly in much the same way that UNIX is user-friendly to those who have already been introduced to the computer world, and are interested in really getting some work done! Thus the particular style chosen is not of the "hold the user's hand", menu-driven variety, nor of the "free to choose anything", full screen editor variety, but one which was designed specifically to be very fast and psychologically painless for the parametric analysis user. In other words, it was designed to relieve the competent user of the overwhelming boredom of the former, while at the same time, providing enough structure so as to eliminate the extra decisions and keystrokes which are usually associated with the latter, which could certainly lead to undue fatique during an involved parametric analysis study. The fact that the result is alittle terse as compared to these alternatives does not really bother me in the least, because MIT students will get familiar with the this style soon enough. It gives exactly the same sort of "feel" as UNIX does, and in my opinion in the long run- which is only a short haul from here, this particular style chosen will have turned out to be a real blessing.

Thus the user-interfaces of SOLPAS and CALPAS1 are not encumbered with the typical gaudiness that I associate with most software products such as word processors and other canned software packages that have been marketed with a pretty wrapping. Not only does this interface appear much cleaner in nature, it also consequentially allows for it's "plug-in, plug-out" capability. The result is that screen images put forth a straightforward, no frills, no nonsense image, as the reader will gather by looking at the Appendix, where he or she will find an example of the input and output produced by these modules. Questions are asked one at a time in a structured fashion, presenting the current default option, which the user is free to quickly accept via the carriage-return, or to reject by writing over the default with his own answer. The user's response is checked by the interface, and rejected if it does not fall within allowable outer limits. If this is the case, a simple error message appears beside the answer on the same line, telling the user what the allowable outer limits are. When the user finally gives a reasonable response, the error message disappears, the answer is accepted, and the interface moves on to the next question. In most situations assuming this software will in fact be used for parametric analysis- as opposed to previous building systems software, the user will for the most part only need to continually hit the carriage-return to enter most of the input. While this is an added task not needed with the full-screen editor variety of user-interface, it has the advantage of requiring the user to quickly check each default option, making sure it is indeed appropriate before proceeding. Thus this feature was inaugerated intentionally to save time in the long run, due to it's tendency to alleviate careless input errors. All input is logically organized into screens which have logically related data elements. Thus if one pulls in an existing file to reedit for parametric analysis purposes, the user needs only to look at the screens which contain the pertinent data to be varied. Thus a minimum of decisions and keystrokes are required which in my mind, bring together the best of both of the alternative methods mentioned above, and the system is quite fast. It normally only takes a matter of seconds to call up the one or two screens, and change the one or two items which are to be varied for the next parametric study, before the system is off and running again.

I should backtrack for a second and mention my use of program input defaults. Since the reader remembers my earlier rhetoric on the subject, he or she is no doubt wondering by now why I included them, and if they have the same purpose as they have traditionally served in building systems software. My purpose for including input defaults was twofold. First, given the fact that this software is to be used in a parametric analysis fashion, there will normally be an input default for every screen item- which is simply the value from the last parametric run. Secondly, high and low limits are used so that a designer can "set the environment" for parametric analysis- so to speak, so that things will go faster, and he or she doesn't have to be totally conscious every moment to make sure that various heat transfer coefficients, etc.... which won't normally be varied from moment to moment, stay in bounds so as not to throw the results off. Thus these limits are meant to be set at the beginning of a project which will undergo parametric analysis. To me, this is a valuable feature, and is a far cry from the traditional use of defaults whereby they are used solely to save the user some inputting effort, letting him put a rather simple building through a "garbage in- garbage out" process. Since I didn't inaugurate a full screen editor type of user-interface which lets the user get away with skipping over vast amounts of input, my default scheme is less apt to be used in this rather useless fashion.

I should also mention that one traditional feature which has not been incorporated into the user-interface scheme at this time, is a "help" facility. Such a facility was not included in the original user-interface components because I didn't want to overencumber them with extraneous code. Instead, I plan to make use of DeSmet's "flip-screen" assembly functions, once I finally get around to adding this facility, as I think it will be a much cleaner approach. Such a facility would not erase a screen to display background information regarding input data at the user's request. Instead, at the touch of a key, the text page in memory which is sent to the CRT would be instantaneously "flipped" to display a new page which has this information, then at a second touch of the key, the user would find himself instantaneously back where he left off. This is a much faster approach than the traditional BASIC method, and also leads to further code modularity. In addition, if one already has the assembly routines available, it should in general be much easier to implement.

In terms of performance and ease of overall use, a subsequent parametric recalculation takes only a second or two to complete before the new SOLPAS results replace the old results on the CRT. Up to four runs can then be graphed together using LOTUS's spreadsheet graphing facilities, allowing one to compare a number of slightly modified building designs in a particular city, or to see how the same scheme would perform in various cities. All the graphing is handled automatically via LOTUS macros, all that remains for the user to do is to specify how many situations are to be graphed at once- a pretty painless process. Hitting the carriage-return after viewing the last graph, will automatically put the user back in the SOLPAS module, giving he or she the options of archiving the last run and/or doing another one, until the module is finally exited. All in all, the process of using SOLPAS is quite fast, requiring a minimum amount of learning effort. In addition, a utility called SHOW is available which gives the user the opportunity to peruse weather files for any of the cities available, or any previously stored input files, as well as information pertaining to the assumptions inherent in the 94 different kinds of passive systems. A MAKEFILE utility is also included for SOLPAS which lets the user create new weather files for the system from the reference information available on the SLR method (1).

Because CALPAS1 takes on the order of 25 minutes to run, the sequence involved is alittle bit different. Although the input screens are set up in exactly the same fashion as SOLPAS input screens, a calculation is not automatically done following the completion of program input. Instead, you have the option to run a simulation on the input file you've just created, or go back and immediately build other variations on your original theme. These building description files are placed in a batch que to be processed all at once, alleviating the need for the user to sit there waiting for one process to complete itself before he or she starts the next process. The whole idea relates back to my scheme in an earlier chapter where an architect at this stage of the design process, since his options are probably more limited, can effectively try out a few variations at once and come back later when they are finished to check out how the results differ. There's really no reason for him to sit around and wait, and the batch que can currently be loaded up with as many as ten variations, which could even be left to run overnight. Since the CALPAS1 Output Processor is completely decoupled from both the input and calculation sections as described earlier, the results of any design variations run can be called up, looked at, and graphed via LOTUS 123 at any later time. Any data that CALPAS1 puts out can be graphed, including space & storage temperatures during any given hour of the year, and heating, venting & cooling tallies on an hourly, daily, or monthly basis. In addition, the Output Processor will even graph yearly summary results between various CALPAS1 runs, which at a glance gives the user an indication as to which of his design variations is the most energy efficient. Via the method described in the previous chapter, in the finished product, output data from various runs will be able to be intermixed, tabularized, and graphed in a wide variety of ways, and the user will be able to specify that he or she wants multiple ranges of detailed output. While CALPAS1 isn't a fast module to run, all of these features make the time delay involved easier to live with, and the result is a marked improvement over the original "bare-bones" program.

To date, this software system has not undergone any substantial use, which is always the true grit test. The links between the various modules were rather slow in evolving due to the need to get this thesis written. Consequently, our original hope of testing out the finished product in this semester's building systems courses, has not materialized. Therefore I can not offer the reader any substantial proof other than my own experimentations as it was being developed, as to whether the system actually works as anticipated in an educational setting or not. I can however reiterate the fact that what has been written is adaptable to change, if this is found to be necessary at a later date. For instance, if my rather terse user-interface is found to be too terse for beginning computer users, it can rather easily be replaced with the "hold your hand" variety, by substituting components which work in this manner. Since a prototypical software system is by it's very nature experimental, I have no doubt that changes will be required down the road, as new and previously unanticipated needs arise.

As I have mentioned, this software is still under development and probably will be throughout the summer. Next fall's MIT students should be able to use the finished product in their building technology courses. Aside from the improvements that I have already mentioned, a final enhancement I would like to make, is to do away with LOTUS 123 as a graphical output tool. While LOTUS, being written in assembly code does indeed throw graphs up on the screen fast, here is where the system as it stands now looses it's coherence. Calling up 123.EXE is indeed awkward, and the process of having to use macros to get the data in and the graphs up is slow, and not a very clean approach. If I were to write my own business graphics routines of which algorithyms are readily available, the actual process of getting the graphs drawn would be slower, but the whole process would be transparent, and would not adversely affect the overall coherence of the software system as it does now. As the system stands now, this is my greatest complain, and one which I will surely address when I finally get the time-which will probably be when I retire! The only other major side experiment on my agenda, would be to experiment with a local database program which is callable in C, to see if it can be used in an efficient and coherent fashion as a file archiving device. If it can, this obviously will open up some new avenues to explore. Otherwise, if coherence and efficiency can not be maintained so that the overall effect is similar to using LOTUS 123 as a graphical output tool, exploring this direction is in my mind, not worth it.

Other than this, the system is coherent at both the user, and the software maintenance levels, and it's algorithyms are appropriately matched with the phases of the architectural design process in which it will be used. The algorithyms were also very carefully chosen so as to be able to work together well- to compliment the strengths and weaknesses of each other so as to provide a comprehensive design tool which remains useful throughout the full range of the architectural design process. This should go a long ways toward making this system a useful software development package for training students in the fundamentals of passive solar design. The fact that the system was built to communicate a graphical representation of output results, as well as to facilitate efficient parametric analysis, should make this design tool a much more effective educational tool than it's predecessors in fact were, as the reader saw in the first chapter of this thesis.

After a final evaluation by the Athena flagship, and a final "tune-up" based on the recommendations received, this software will be placed on-line on the Project Athena system here at MIT, where the true test of time will tell whether my hypotheses, and the subsequent coding effort involved were worth anything at all. Since as yet, no substantive Athena software has been written in this department using the "flagship- big guns approach", although a few potshots have been taken here and there, the real battle between these opposing aggressors will not take place for quite awhile. Given the fact that Athena has had lots of trouble getting it's big guns in place, the waiting period could be endlessly prolonged- to the point where the battle never takes place at all!

In the meantime, if my software actually gets used by a fair number of students, and is maintained by those who may be intrigued enough by it to want to experiment with the code and add new features, I will consider this project highly successful, as this will be a real " first" for a software development project in the field of building energy analysis, here within the architecture school at MIT.

CHAPTER 6: CONCLUSION

"Therefore, my beloved brethren, be steadfast, immovable, always abounding in the work of the Lord, knowing that your toil is not in vain, in the Lord."

Without reiterating half of this thesis, I would like to say some concluding remarks regarding this ongoing experiment, and how it has affected my thoughts on software development, and in particular, the design of educational software. If I have anything at all to conclude from my experiences using and developing design tools, it is this: Design tool software must enhance creativity- not stifle it! Or else even if it does crank out numbers which have some usefulness, the software will still be of little value and will consequently be relegated to the back shelf! This overall objective seems directly tied to the fact that architectural design is a so-called "wicked design problem" (1), therefore whatever particular design problem is at hand needs to be explored from various angles since there is no "right answer" to such a design problem, nor is there a "right way of getting there". To this end, a design tool requires good parametric design capabilities, not only allowing- but actually encouraging experimentation, so as to be able to teach the user something that he or she did not already know, thus opening up avenues for design exploration. This overriding need has many implications.

First, the number one quality a design tool must have in my mind, is flexibility, because without it- it can do little to foster creativity. Of course it must have other qualities as well- such as efficiency and "user-friendliness", or else it will not be useful at all. These really basic needs have been addressed before in earlier design tools, as I mentioned in the earlier chapters, but in general these early tools fell way short in terms of their primary educational need- which is a high level of built-in flexibility which can accomodate the tool's main objective, which is to foster design creativity in those areas of design where a calculation tool can be beneficial.

It is generally acknowledged that UNIX/C is currently the most popular operating system/development environment for software development (2), to those who are engaged in this highly creative endeavor day by day, and this has led me to ask the question: Why? It seems that there ought to be some fairly strong implications here. If the UNIX/C environment is successful in providing a user environment which has the qualities which foster creative software development, then it seems natural to deduce that these external characteristics are simply an outward manifestation of it's internal structure, which has in it enough integrity to make it's true nature show through on the surface. I am no expert in linguistic theory, but it makes common sense to me that if a language has a certain consistency on the inside, that this will make it have a particular consistency on the outside as well.

Early on in this project, I discovered the power and flexibility of the C programming language, that in fact C provided a highly creative software development environment. In essence, the closer I looked at C, the more I saw that C had the exact qualities about it, that a good design tool in an educational environment needed. Since these qualities have been spoken about in great detail already, I will not reiterate them here. But this awareness combined with the above common sense hypothesis, led me to think that if I used C as my programming environment, and as much as possible, emulated the internal structure of this language in my own software, by utilizing it's full power and expression, then my software would be able to take on these same characteristics itself. This is why I am so adamant in my assertion that existing code should not be simply directly translated into C from another languages such as BASIC or FORTRAN. To do so is a grave injustice because the result will still most likely emulate the flavor of these languages, and not take on the pristine qualities of C.

This in turn guided me in choosing a software system development philosophy which seemed to be the most logically consistent with the internal structure of the UNIX/C environment, so that the qualities in this environment that I wanted to see in my own applications software system would not be degraded, but actually built-up to an even greater degree.

The resulting "cruise missile approach", was so consistent on a conceptual level, with both my design tool goals, and the overall software system that I was working under, that I had no trouble at all turning my software design theory into actual implemented code. The end result being that the underlying qualities I wanted in my design tool system to enhance creativity, (and prolong the useful life of the software, promoting future growth)- actually do I think show through on the surface, just as I had postulated.

I do not think the results would be anywhere near as dramatic had this internal consistency not be maintained between the overall goals for the software, the software design approach taken, and the software used to produce it. Had I been forced into using Athena's "flagship- big guns" software design approach, or through constraints had been forced to use one of the more traditional scientific & engineering computer languages, I think the results would suffer the inevitable consequence- i.e.- the resulting software would be inhibited from fully expressing the qualities that were originally sought in it's design. In this regard, designing a large piece of software, is just like designing a large building, and software designer's would do well I think, to study the designs of this centuries more famous designers- whether it be the industrial designs of Charles Eames, or the architectural masterpieces of someone like Frank Lloyd Wright. One would soon discover after looking over the design endeavors these men undertook, that design expression on the outside fundamentally relates to how design components are used together, and what material qualities these components lend to the greater scheme of things- as Louis Khan has so eloquently written about (3). (Notice that I am not referring the reader to any "post-modern" architectural designers such as Venturi or Graves!)

Being aware of this fundamental design axiom through my previous architectural training, I began looking for the right material- i.e.- the right generic components of which modular, reusable building elements could be fashioned that could then be used to build up my software system according to the plan I had conceptualized. Making sure that the components used had the right integral qualities so as to enhance and preserve the qualities desired in the final built product. Hence all my efforts to develop a library of succinct, ascii C functions which could be used over and over again, providing the essential link between the various algorithyms of the different modules, and the text-oriented, outside world of the UNIX environment. The internal integrity afforded through the proper choice of components I believe, has lent the whole system a real degree of coherence on the outside, both from the point of view of the user, and of the programmer who must maintain the system.

In choosing these components, just like in choosing the proper building elements for a building, the question of the optimal size of these components can not be overlooked. In other words, the proper scale of these elements is important.

Here's where an understanding of modular building theory came into the picture. My analogy of a geodesic dome was carefully chosen, to illustrate the flexibility and strength afforded as a result of generic components which are sized properly in relation to each other. This required a real dedication to true K & R C-coding philosophy where compact functions are built up via other svelte functions, creating a multivalent effect which gives the overall system a "tinker-toy" capacity to evolve in multiple ways, providing the sought qualities of flexibility and tenacity.

An additional by-product of my adherence to modular building theory, was the opportunity to pursue a high level of code integration, which made the coding of these rather large modules much easier, because many small pieces of code could be logically grouped together and segregated off into outside source files. This dramatically "cleaned-up" the local environment, leaving the module algorithyms intact and unencumbered by extraneous garbage. Which- just like modular buildings, makes the system code easily maintainable, and gives it a "plug-in, plug-out" nature, allowing for a high degree of adaptation and future expansive growth.

At this point, only time will tell, but I do believe that through this experimentation, I have discovered some valuable insights on how to go about designing and coding software which must have certain qualities to be able to survive in an educational setting. (In my mind, the phrases "educational setting" and "design setting" are synonymous.) After spending the last several years using and maintaining architectural design tool software, I also believe that the code I have written on a qualitative level, is much different from most of the code that has been written before it, for all of the reasons and implications that are stated above. And had I not discovered the C language when I did, just like my predecessors, what I have done would not have been possible to do.

Through my trials and errors, I know that for myself, I have discovered what I consider to be a bonafide software design theory, which is the conceptual process that I have outlined above, as seen in the context of how it evolved- which is what the main portion of this thesis has tried to explain. It is not based on the traditional linear theory of "top-down design", but is much closer in nature to the more cyclic and convoluted design process that architectural design is (1). While it does on the largest level encompass a "from big to small" approach- as does "top-down design", it also concurrently operates on other levels which require a "from the bottom up" thinking process as well, in order to maintain design integrity from small to big. This design integrity is of course essential in being able to achieve one's overall software design goals. This process is indeed "wicked" in that in any given moment, where the design process leads next is not entirely predictable (to the creative designer). There is always more than one avenue to choose from in order to head in a given direction, and usually at least at the onset, more than one direction to go in. While the procedure I have outlined above was indeed what I actually did, for the sake of explaining it in a logical fashion, I have "straightened it out"- as it did not actually occur in the straightforward manner that I described it in, but in the more architectural design fashion of working at one end of the spectrum for a little while, then shifting to the other end. (If the reader has found this thesis alittle hard to follow at some points, since it was written chronologically as a journal, it only goes to show what I am now trying to point out.) This was not a conscious procedure on my part, it just happened to be the way things needed to happen, as experiments at one end of the spectrum would open up new ideas at the other end. In my mind, such a design process is essential in order to produce "good software", just as it is essential in designing a "good building"! And just like designing a building in the real-world, whether or not you have the resources and time on your hands in order to be able to do this is the key factor. Probably not!, which makes coming up with a personal design philosophy which seems to work well, all the more important as it gives you that needed edge- i.e.- bag of tricks which can save you alot of time on your next job.

For an overall evaluation of my experience to date, although this project is still ongoing (and due to the fact that I have that true designer's spirit, forever will be in one way or another), I consider this thesis closed, because the overall goals that I have stated for it in the introduction, I believe have been fulfilled. The most important aspect I think a designer needs to gain from his or her education, is to be able to come up with a personal design philosophy of one sort or another at the end. While such a philosophy hopefully will grow and may even evolve into something quite abit different later, at least it is a solid foundation to build on for the time being. Again, my second goal is that this work will provide in one way or another, a big inspiration for others to try to get their "sea legs" as well. Even if they don't agree with me and would rather carry a swiss army knife around in their pocket- rather than learning how to row, at least this second goal will have been met as well.

CHAPTER NOTES

Introduction

1. Krinkle, David L., Integration of Energy Analyses in Design Through the Use of Microcomputers, Masters Thesis, Massachusetts Institute of Technology, 1983, p. 19.

2. Sutherland, I.E., Sketchpad: A Man-made Graphical Communication System, MIT Lincoln Lab Tech. Rep. 296, May 1965.

Chapter 0

1. Kohler, Joseph T., Douglas E. Mahone, and Paul W. Sullivan, Pascalc II, Total Environmental Action, Inc., Harrisville, NH, 1980.

2. Kohler, Joseph T., and Paul W. Sullivan, TEANET User's Manual, Total Environmental Action, Inc., Harrisville, NH, 1979.

3. CALPAS3 User Manual, Berkeley Solar Group, Berkeley, California, 1982.

4. Balcomb, J. Douglas, et.al., Passive Solar Design Handbook, Volume Two of Two Volumes: Passive Solar Design Analysis, U.S. Department of Energy, 1980, Chap. D.

5. ibid., p. 9.

6. ibid., p. 33.

7. Mazria, Edward, The Passive Solar Energy Handbook, Rodale Press, 1979, Chap. IV.

8. St. Clair, Charles, QUICKPAS: A Microcomputer Based Passive Solar Analytical Design Tool, Masters Thesis, Massachusetts Institute of Technology, 1984, p. 29-31.

9. Krinkle, David L., Integration of Energy Analyses in Design Through the Use of Microcomputers, Masters Thesis, Massachusetts Institute of Technology, 1983.

Chapter 1.

1. California Energy Commission, CALPAS1 Program User's Guide, Sacramento, California, 1981.

2. Schneider, G.M., and S.C. Bruell, Advanced Programming and Problem Solving with Pascal, John Wiley & Sons, New York, 1981.

3. Microcomputer Methods for Solar Design and Analysis, SERI, February 1981, p. 0.

4. DOE-2 Program Manual, Los Alamos Scientific Laboratory, U.S. Department of Energy, 1979.

5. California Office of Appropriate Technology, MICROPAS - A Microcomputer Program for Residential Building Energy Analysis, User's Manual, Sacramento, California, 1982.

6. Wirth, Niklaus, Programming in MODULA-2, Springer-Verlag, New York, NY, 1982.

7. BYTE, Byte Publications, Inc., October, 1983.

8. W.S. Fleming and Associates, ASEAM - A Simplified Energy Analysis Method, Report to the U.S. Department of Energy, Washington, D.C., 1983.

9. Bryan, Harvey J., and Steven E. Lotz, Welcome to the Designers Software Exchange, Lab of Architecture and Urban Planning, Massachusetts Institute of Technology, 1983.

Chapter 2.

1. Skjellum, Anthony: "C Instead of FORTRAN?", Compu. Lang. 2(2):33-40, February, 1985.

2. Burger, Brian H.: "C to Assembly Interface", Compu. Lang. 2(2):50, February, 1985.

3. ibid., p. 49-57.

4. Pressman, Roger S., Software Engineering: A Practitioner's Approach, McGraw-Hill, Inc., 1982, p. 4.

5. Skjellum, Anthony: "C Instead of FORTRAN?", Compu. Lang. 2(2):34, February, 1985.

6. Purdum, Jack J., Timothy C. Leslie, and Alan L. Stegemoller, C Programmer's Library, Que Corporation, Indianapolis, Indianna, 1984, Chap. 2.

Chapter 3.

1. Ferierra, Joseph, Conversation at weekly Athena meeting.

2. ibid., Conversation at weekly Ayhena meeting.

3. Williams, Gregg: "The AT&T UNIX PC", BYTE 10(5):98-105, May, 1985.

4. Fawcette, James E.: "Watch Out JAZZ", Infoworld 7(13):5, April, 1985.

5. ibid.

6. Eastman, C.M. and M. Henrion: "GLIDE: A Language for Designing Information Systems", Computer Graphics 11(2):24, Summer 1977.

7. Yates, Jean L.: "UNIX and the Standardization of Small Computer Systems", BYTE 8(10):161, October, 1983.

8. ibid., p. 210-211.

9. Krieger, Mark and Fred Pack: "UNIX as an Applications Environment", BYTE 8(10):212, October, 1983.

Chapter 4.

1. Bryan, Harvey J. and David Krinkle, "MICROLITE: A Microcomputer Program for Daylighting Design", Proceedings of the Seventh National Passive Solar Conference, Knoxville, Tennessee, September, 1982. p. 405.

2. The Energy Group, TNODE: A Thermal Network Analysis Program for the IBM Personal Computer, Georgia Tech, August, 1984.

3. California Energy Commission, CALPAS1 Program User's Guide, Sacramento, California, 1981.

4. Haley, Robert B., SOLPAS - A Passive Solar Design Program, 1983.

5. Balcomb, J. Douglas, et.al., Passive Solar Design Handbook, Volume Two of Two Volumes: Passive Solar Design Analysis, U.S. Department of Energy, 1980.

6. Morris, W. Scott: "Road Map for Passive Design", Solar Age, May 1983, p. 50.

7. ibid., p. 49.

8. California Energy Commission, Passive Solar Handbook, Sacramento, California, p. 312.

9. Balcomb, J. Douglas, et.al., Passive Solar Design Handbook, Volume Two of Two Volumes: Passive Solar Design Analysis, U.S. Department of Energy, 1980.

10. Krinkle, David L., Integration of Energy Analyses in Design Through the Use of Microcomputers, Masters Thesis, Massachusetts Institute of Technology, 1983, p. 19.

11. Krieger, Mark and Fred Pack: "UNIX as an Applications Environment", BYTE 8(10):212, October, 1983.

12. The AutoCAD 2 Drafting Package User Guide, Autodesk Inc., 1984.

Chapter 5.

1. Balcomb, J. Douglas, et.al., Passive Solar Design Handbook III, U.S. Department of Energy, 1980.

Chapter 6.

1. Krinkle, David L., Integration of Energy Analyses in Design Through the Use of Microcomputers, Masters Thesis, Massachusetts Institute of Technology, 1983, p. 19.

2. Krieger, Mark and Fred Pack: "UNIX as an Applications Environment", BYTE 8(10):210, October, 1983.

3. Lobell, John, Between Silence and Light: Spirit in the Architecture of Louis Kahn, Shambhah, Boulder, Colorado 1979.

NOTE

All chapter opening quotations are from the BIBLE, NASB version, an excellent and practical guide to spiritual living:

> "All Scripture is inspired by God and profitable for teaching, for reproof, for correction, for training in righteousness; that the man of God may be equipped for every good work."
>
> 2 TIM 3:16

References for the chapter opening quotes are as follows:

Chapter 0:	MAT 12:20
Chapter 1:	1 COR 15:37
Chapter 2:	HEB 11:1
Chapter 3:	JAM 2
Chapter 4:	HEB 12:1
Chapter 5:	HEB 12:11
Chapter 6:	1 COR 15:58

A highly recommended book!

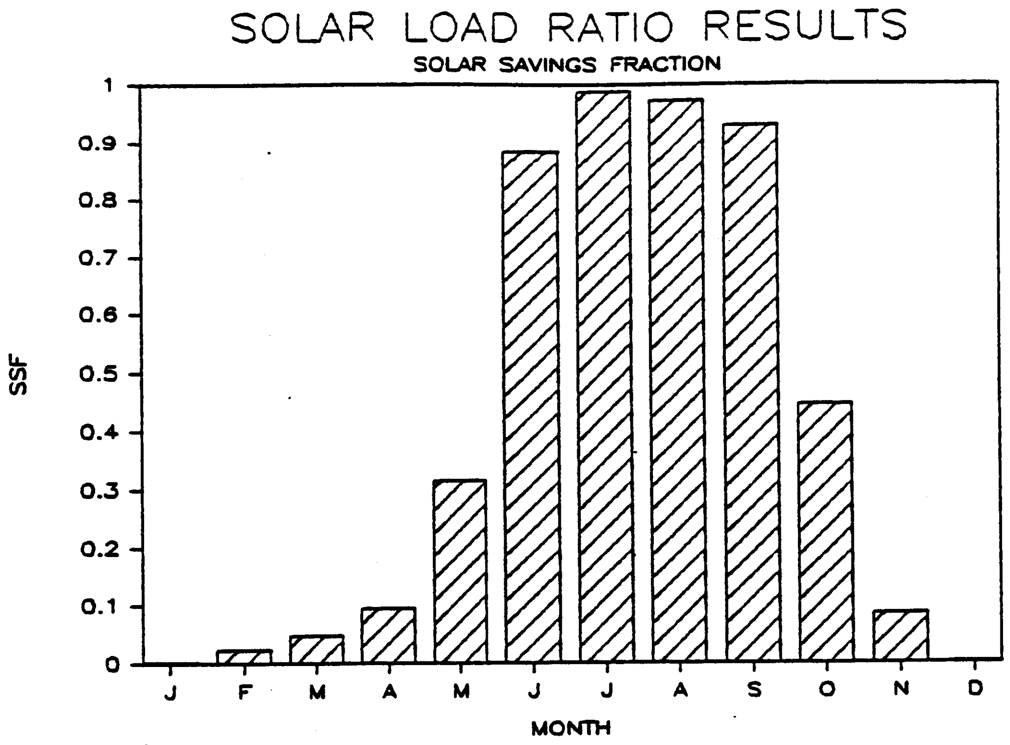
APPENDIX

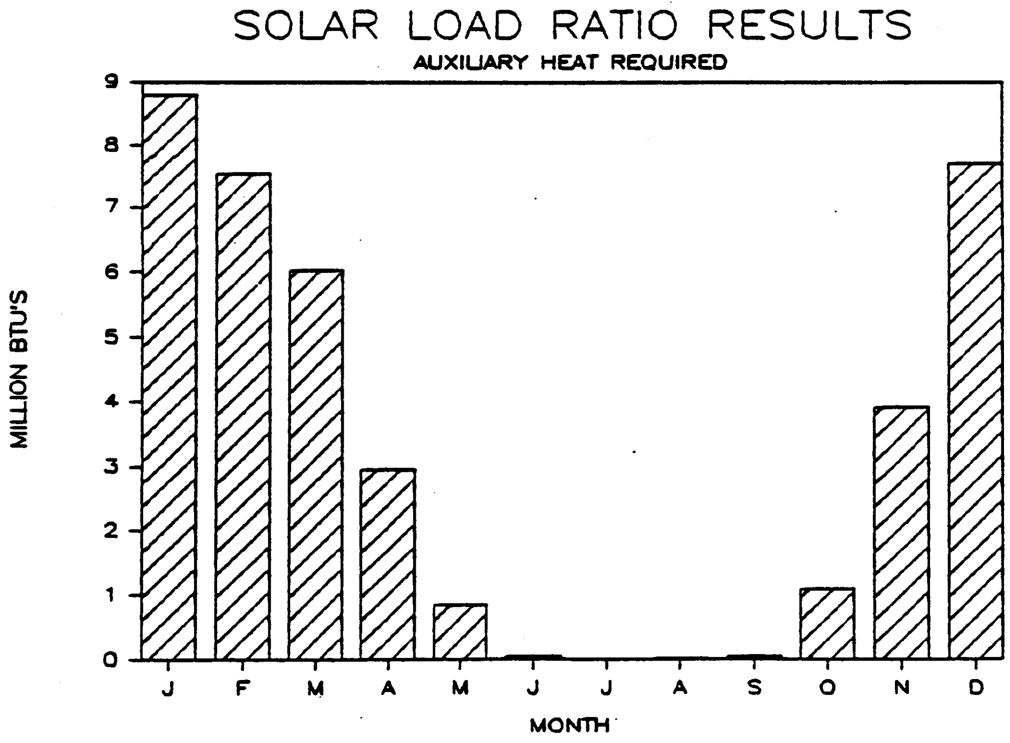
```
You will be asked to enter the dimensions of your building in
to have the program compute the Building Load Coefficient(BLC)
you have already computed the BLC and wish to skip this proces
'skip'.  If you want to use an existing file, either an attrib
or a previously stored blc file- enter 'file'.  Otherwise just
'return' to continue ....

Use an attribute file:  (y,n) ?   n



You may have a building that utilizes more than one of the pas
systems.  For clarity, let us call each system a 'zone'.  You
select from one to three zones.  You will be asked to give the
facing glazing dimensions for each zone.

No. of zones(1-3):       1



                               SOUTH WALL MODULE

South wall height:       9
South wall width:        51
South wall r-value:      12
South wall area:         459

Is all data correct:  (y,n) ?   y



                             DATA FOR ZONE NUMBER 1

System number(1-94):     64
Overhang ratio(x/h):     0.500
Overhang ratio(y/h):     0.250
South glazing height:    4.5
South glazing width:     18.0
South glazing area:      81.0

Total south glass:       81.0
Net south wall area:     378.0

Access east wall module   (y,n)?   y
```

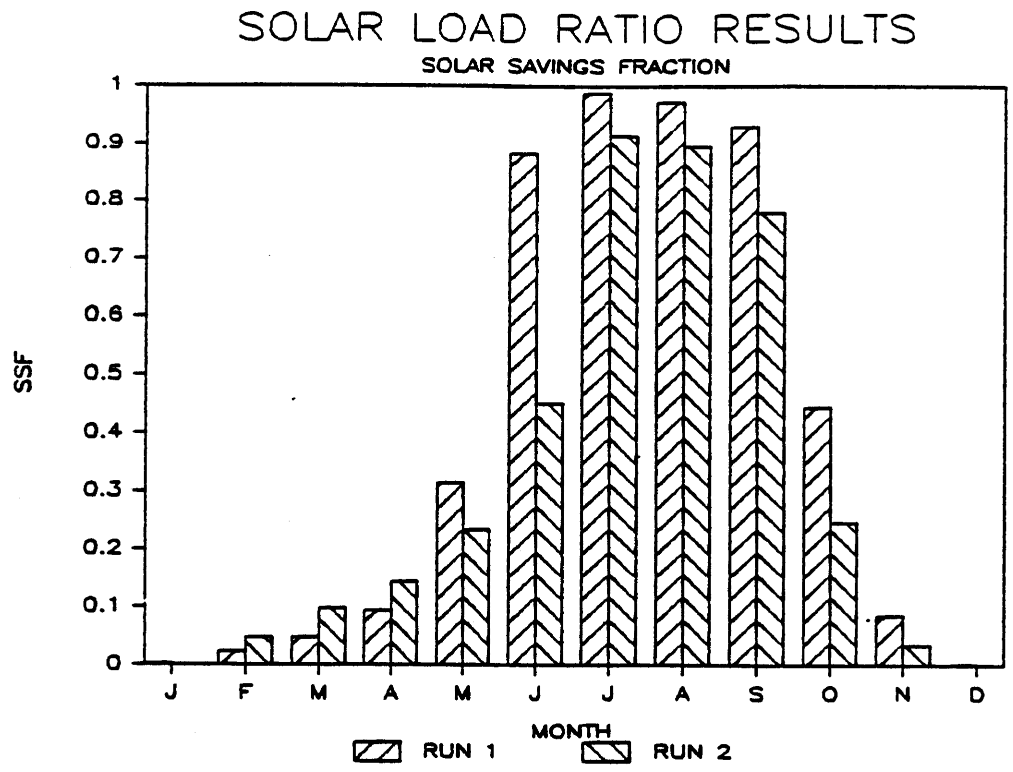
```
                                    EAST WALL MODULE

East wall height:        9
East wall width:         26
East wall r-value:       12
East wall area:          234
East glazing area:       49.0
Number of glazings:      2
Net east wall area:      185.0

Is all data correct:  (y,n) ?   y


                                    WEST WALL MODULE

West wall height:        9
West wall width:         26
West wall r-value:       12
West wall area:          234
West glazing area:       73.0
Number of glazings:      2
Net west wall area:      161.0

Is all data correct:  (y,n) ?   y


                                    NORTH WALL MODULE

North wall height:       9
North wall width:        51
North wall r-value:      12
North wall area:         459
North glazing area:      25.0
Number of glazings:      2
Net north wall area:     434.0

Is all data correct:  (y,n) ?   y


                                      ROOF MODULE

Roof length:             51
Roof width:              26
Roof r-value:            20
Roof area:               1326

Is all data correct:  (y,n) ?   y
```

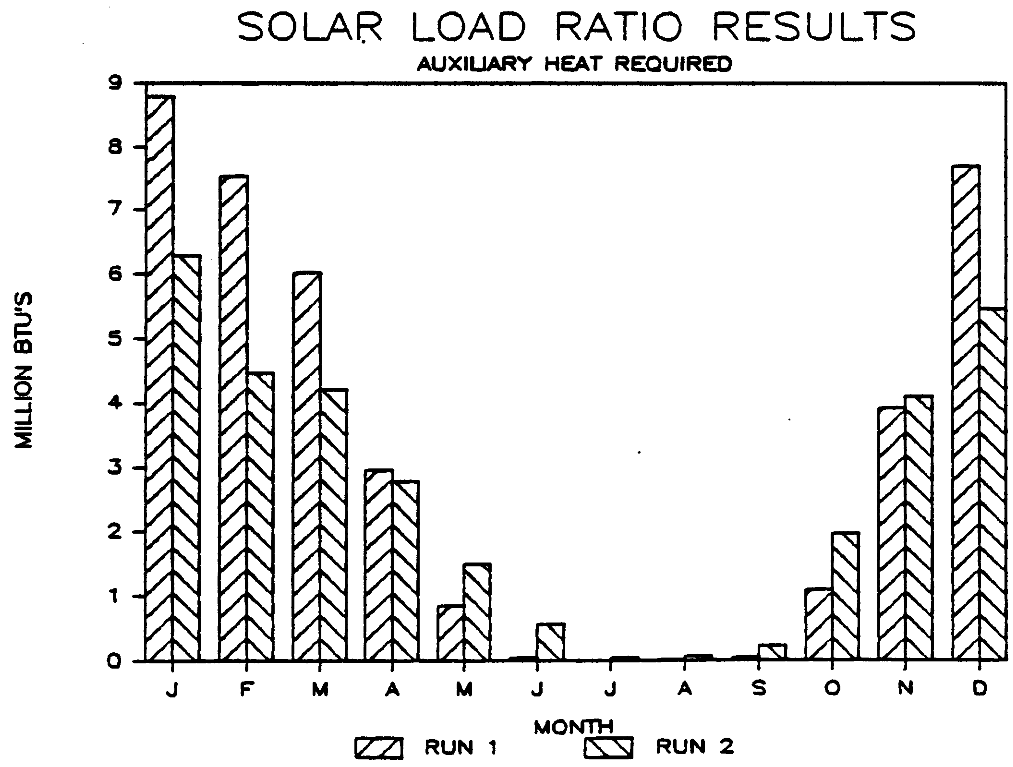
```
                         FOUNDATION MODULE

Foundation:  (s,c,b,<ESC>) ?  s
Perimeter length:       152
Perimeter r-value:      12

Is all data correct:  (y,n) ?   y



            AIR CHANGE,CEILING HEIGHT, AND COMBINED FLOOR AREA

Air changes/hr:         0.5
Ceiling height:         8
Combined floor area:    1250

Is all data correct:  (y,n) ?   y



                  BLC CALCULATION

Blc:                    8872
Lcr:                    109

Access thermostat module  (y,n) ?   y



                  THERMOSTAT SET-POINT AND INTERNAL GAINS MODUL

thermostat setting:     65
Internal gains:         40000

Is all data correct:  (y,n) ?   y
```

```
SOLPAS1                  SOLAR SAVINGS            AUXILIARY HEATING
BOSTON                     FRACTION                  REQUIREMENT

january                     0.001                    8788396   Btu
february                    0.023                    7520579   Btu
march                       0.047                    6028486   Btu
april                       0.095                    2965615   Btu
may                         0.316                     852605   Btu
june                        0.883                      43711   Btu
july                        0.987                       1934   Btu
august                      0.972                       7049   Btu
september                   0.929                      38722   Btu
october                     0.446                    1098814   Btu
november                    0.086                    3916398   Btu
december                    0.001                    7691378   Btu

ANNUAL                      0.063                   38953686   Btu

ANNUAL ENERGY SAVINGS OVER
REFERENCE NON-SOLAR BUILDING                         3550762   Btu

Want a hardcopy:  (y,n) ?   n
```


SOLAR LOAD RATIO RESULTS
SOLAR SAVINGS FRACTION
SSF
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
J
F
M
A
M
J
J
A
S
O
N
D
MONTH


SOLAR LOAD RATIO RESULTS
AUXILIARY HEAT REQUIRED
MILLION BTU'S
9
8
7
6
5
4
3
2
1
0
J
F
M
A
M
J
J
A
S
O
N
D
MONTH


SOLAR LOAD RATIO RESULTS
SOLAR SAVINGS FRACTION
SSF
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
J
F
M
A
M
J
J
A
S
O
N
D
MONTH
RUN 1
RUN 2


SOLAR LOAD RATIO RESULTS
AUXILIARY HEAT REQUIRED
MILLION BTU'S
9
8
7
6
5
4
3
2
1
0
J
F
M
A
M
J
J
A
S
O
N
D
MONTH
RUN 1
RUN 2

```
You will be asked to enter information concerning the building
wish to have an energy analysis performed on by Calpas, as wel
other data pertaining to the building's location and operating
schedule.  If you have already been through this process and w
to use an existing building description file that you have pre
stored, or you want to enter data from an attribute file creat
another program, enter 'file', otherwise just press 'return' t
continue .....

Use an attribute file  (y,n)?   n



                              RUN DATA MODULE

first month of run:                    jan
last month of run:                     jan
first month of summer mode:            jun
last month of summer mode:             jun

Is all this data correct  (y,n)?       y



                           OUTPUT OPTIONS MODULE

skip monthly output  (y,n)?            n
skip daily output  (y,n)?              n
skip hourly output  (y,n)?             n
first day of daily output:             jan 17
last day of daily output:              jan 17
first hour of hourly output:           1am
last hour of hourly output:            12pm

Is all this data correct  (y,n)?       y



                           LOCATION DATA MODULE

latitude of site:                      37.70
azimuth of south wall:                 0.00
mean temp of month preceeding run:     49.90
mean ground reflect. (e,w,s):          0.20
initial air & storage temps. :         65.00
total internal heat gain/day:          68260.00

Is all this data correct  (y,n)?       y
```

```
                          BUILDING ENVELOPE AREAS/VOLUME

area of north wall  (opaque):          269.50
area of east wall  (opaque):           258.25
area of west wall  (opaque):           249.50
area of south wall  (opaque):          219.50
area of roof:                          711.00

area of north glazing:                 8.75
area of east glazing:                  8.75
area of west glazing:                  17.50
area of south glazing:                 206.00

volume of conditioned space:           9934.00

Is all this data correct  (y,n)?       y



                        BUILDING ENVELOPE SURFACE DATA MODULE

ave u-value of all exterior walls:     0.05
u-value of roof/ceiling:               0.03
heat loss multiplier for infilt:       89.40
heat loss multiplier for slab:         97.90
mean u-value of nonsouth glazing:      0.58
u-value of direct-gain glazing:        0.10
ave absorbtance of walls & roof:       0.67

trans. of n,e,w glaz. in summer:       0.76
trans. of n,e,w glaz. in winter:       0.78
trans. of south glaz. in summer:       0.32
trans. of south glaz. in winter:       0.43

Is all this data correct  (y,n)?       y



                        INTERNAL HEAT GAIN ABSORBTION MODULE

% of int. gain to air:                 1.00
% of int. heat to s.side m.wall:       0.00
% of int. heat to n.side m.wall:       0.00
% of int. heat to slab/floor:          0.00
% of int. heat to quickwall:           0.00

Is all this data correct  (y,n)?       y
```

```
                    SOUTH SOLAR HEAT GAIN ABSORBTION MODULE

% of south sum sun to air:                0.30
% of south sum sun to s.side m.wall:      0.20
% of south sum sun to slab/floor:         0.50
% of south sum sun to quickwall:          0.00

% of south win sun to air:                0.30
% of south win sun to s.side m.wall:      0.20
% of south win sun to slab/floor:         0.50
% of south win sun to quickwall:          0.00

Is all this data correct  (y,n)?          y
```

```
                   NON-SOUTH SOLAR HEAT GAIN ABSORBTION MODULE

% of non-s sum sun to air:                0.30
% of non-s sum sun to s.side m.wall:      0.10
% of non-s sum sun to n.side m.wall:      0.10
% of non-s sum sun to slab/floor:         0.50
% of non-s sum sun to quickwall:          0.00

% of non-s win sun to air:                0.30
% of non-s win sun to s.side m.wall:      0.10
% of non-s win sun to n.side m.wall:      0.10
% of non-s win sun to slab/floor:         0.50
% of non-s win sun to quickwall:          0.00

Is all this data correct  (y,n)?          y
```

```
                         VENTILATION DATA MODULE

low elev. operable vent. area:            38.50
high elev. operable vent. area:           18.00
elev diff. between low & high vent:       8.50
azimuth of low elev. vent. area:          0.00
wind correction factor:                   0.25
forced vent. fan energy:                  0.00
airchg/hr due to forced ventilation:      1.00
do you want wind ventilation  (y,n)?      n

Is all this data correct  (y,n)?          y
```

THERMOSTAT SETPOINTS MODULE

```
cooling thermo. setpnt:                   80.00
temp setpnt beg/end vent cool mode:       65.00
temp setpnt beg/end vent. heat mode:      80.00
heating thermostat setpnt:                65.00
temp diff. req'd for forced vent:         0.00

Is all this data correct  (y,n)?          y
```

MASS WALL STORAGE DATA MODULE

```
area of s. side m.wall:                   0.00
air film cond. of s. side m.wall:         0.00
air film cond. of n. side m.wall:         0.00
thickness of m.wall  (in):                1.00
vol. heat capacity of m.wall:             1.00
conductivity of m.wall (ft):              1.00
u-value of m.wall glazing:                0.10
rad. heat trans betw glaz. & m.wall:      0.00
air film cond. betw glaz. & room air:     1.50
u-value of m.wall to outside air:         0.00

Is all this data correct  (y,n)?          y
```

SLAB/FLOOR STORAGE DATA MODULE

```
area of slab/floor:                       992.00
air film cond. of slab/floor:             1.10
thickness of slab/floor (in):             4.00
vol. heat capacity of slab/floor:         30.80
conductivity of slab/floor (ft):          9.00

Is all this data correct  (y,n)?          y
```

```
                              QUICK WALL STORAGE DATA MODULE

area of s. side of quickwall:           0.00
air film cond. s. side of quickwall:    0.00
thickness of quickwall (in):            1.00
vol. heat cap. of quickwall:            1.00

vol. heat cap. of light materials:      2488.00

Is all this data correct  (y,n)?        y


                              SOLAR SHADING/INSULATION MODULE

mean hght. of south glazing:            4.50
horiz dist. of ovrhg from s. glazing:   1.50
vert. dist. betw. s. glazing & ovrhg:   0.67
r-value of mov. insul., s. glazing:     0.00

. . . .  running CALPAS 1.0
 bldg desc. file:  lakin2
 weather file:     a:oakland
```

```
                    CALPAS OUTPUT.SELECTION


HOURLY     DAILY      MONTHLY    YEARLY     SUMMARY    QUIT

dates:  jan 17



                       CALPAS HOURLY OUTPUT
                    SPACE & STORAGE TEMPERATURES

HOUR OF        AIR          WALL         FLOOR        QUICKWALL
    DAY        TEMPC        TEMP         TEMP         TEMP

    2am        68.8         70.9         65.0         65.0
    4am        67.4         70.0         65.0         65.0
    6am        66.3         69.0         65.0         65.0
    8am        66.5         68.2         65.0         65.0
   10am        68.9         69.0         65.0         65.0
   12am        72.1         70.7         65.0         65.0
    2pm        74.3         72.7         65.0         65.0
    4pm        75.1         73.9         65.0         65.0
    6pm        74.3         73.6         65.0         65.0
    8pm        73.4         73.2         65.0         65.0
   10pm        72.5         72.7         65.0         65.0
   12pm        71.0         72.0         65.0         65.0



                             CALPAS HOURLY OUTPUT
                               ALL TEMPERATURES

HOUR OF     AMBIENT      AIR          AIR          FLOOR
  DAY        TEMP        TEMP         TEMP         TEMP

  2am        38.0        68.8         68.8         70.9
  4am        35.0        67.4         67.4         70.0
  6am        35.0        66.3         66.3         69.0
  8am        37.0        66.5         66.5         68.2
 10am        40.0        68.9         68.9         69.0
 12am        47.0        72.1         72.1         70.7
  2pm        50.0        74.3         74.3         72.7
  4pm        52.0        75.1         75.1         73.9
  6pm        51.0        74.3         74.3         73.6
  8pm        49.0        73.4         73.4         73.2
 10pm        48.0        72.5         72.5         72.7
 12pm        44.0        71.0         71.0         72.0
```

CALPAS DAILY OUTPUT
ENERGY FLOWS

DAY OF YEAR	HEATING KBTU'S	VENTING KBTU'S	COOLING KBTU'S
"jan 17"	92.00	25.00	15.00
"apr 4"	67.00	46.00	35.00
"may 18"	35.00	87.00	76.00
"oct 12"	43.00	55.00	58.00

CALPAS DAILY OUTPUT
AVERAGE TEMPERATURES

DAY OF YEAR	AMBIENT AVE	AIR AVE	FLOOR AVE	WALL AVE
"jan 17"	23.50	68.36	71.30	65.00
"apr 4"	55.70	72.86	77.20	72.40
"may 18"	73.20	77.86	81.30	77.70
"oct 12"	43.50	74.86	73.80	67.30

CALPAS MONTHLY OUTPUT
ENERGY FLOWS

MONTH OF YEAR	HEATING KBTU'S	VENTING KBTU'S	COOLING KBTU'S
january	100	223	23
february	9	0	0
march	0	18	3
april	2	1076	79
may	0	1635	14
june	0	1576	179
july	0	1788	3
august	0	1558	219
september	0	1676	152
october	0	1706	68
november	0	1505	0
december	37	3	0

CALPAS MONTHLY OUTPUT
HEATING & COOLING PEAK TEMPERATURES

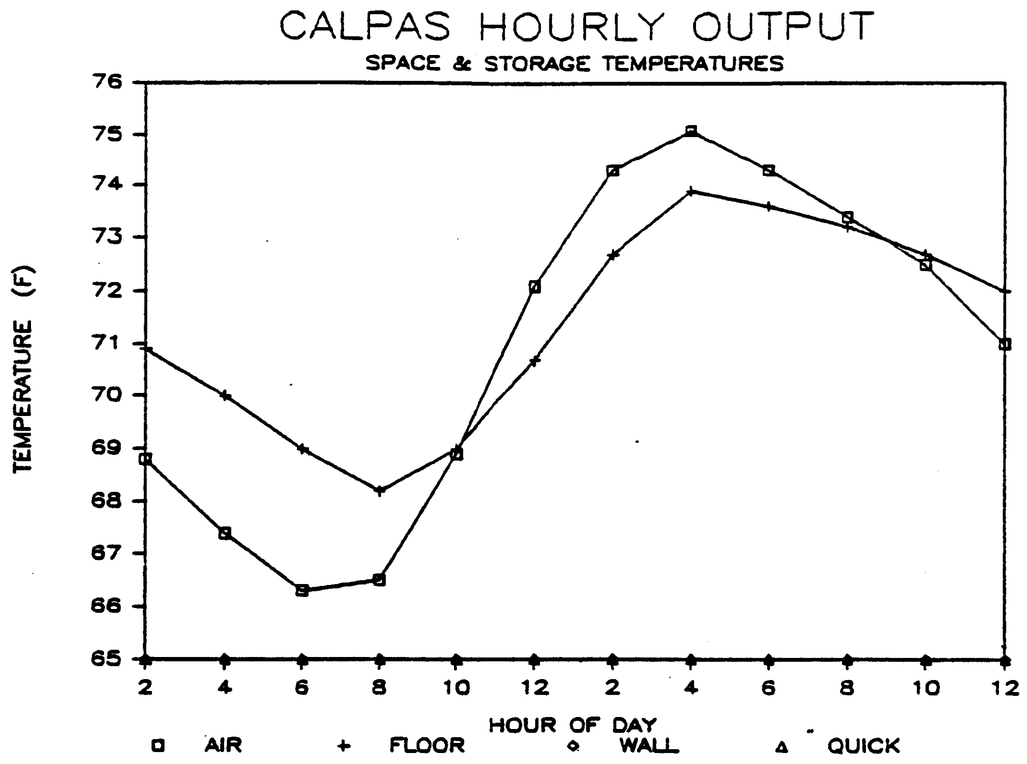
MONTH OF YEAR	HEATING PEAK	COOLING PEAK
january	5026	3819
february	1827	0
march	0	2674
april	804	4861
may	0	3574
june	0	8191
july	0	1851
august	0	6409
september	0	7408
october	0	7049
november	44	0
december	2741	0

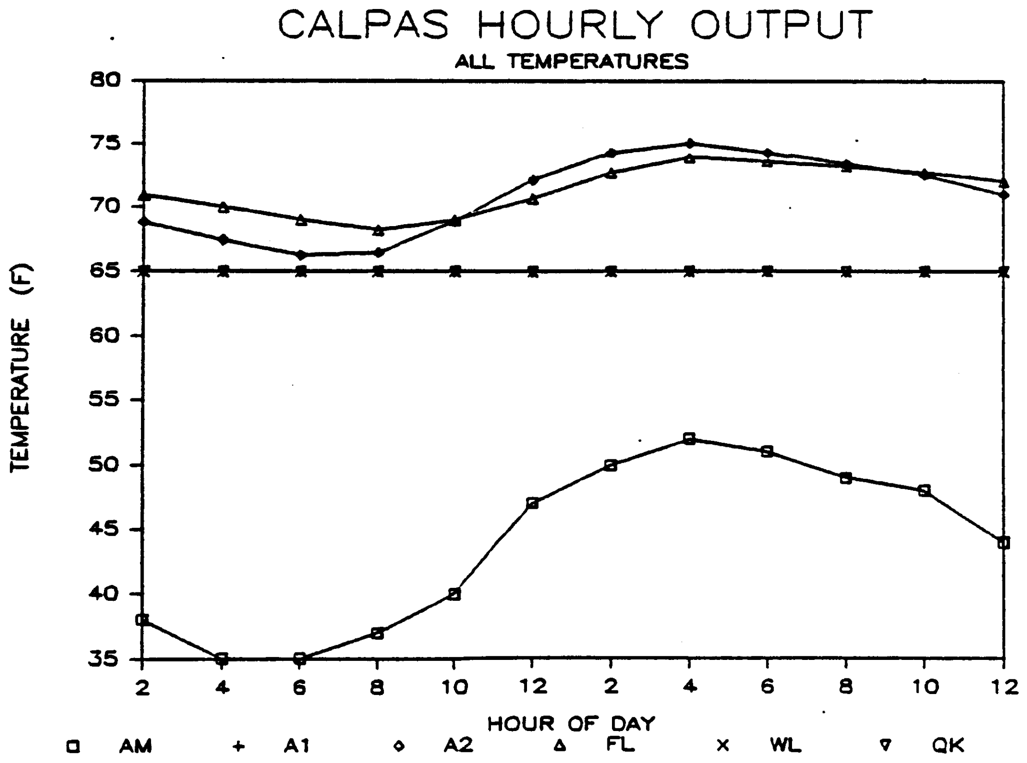
CALPAS YEARLY OUTPUT
ENERGY FLOWS

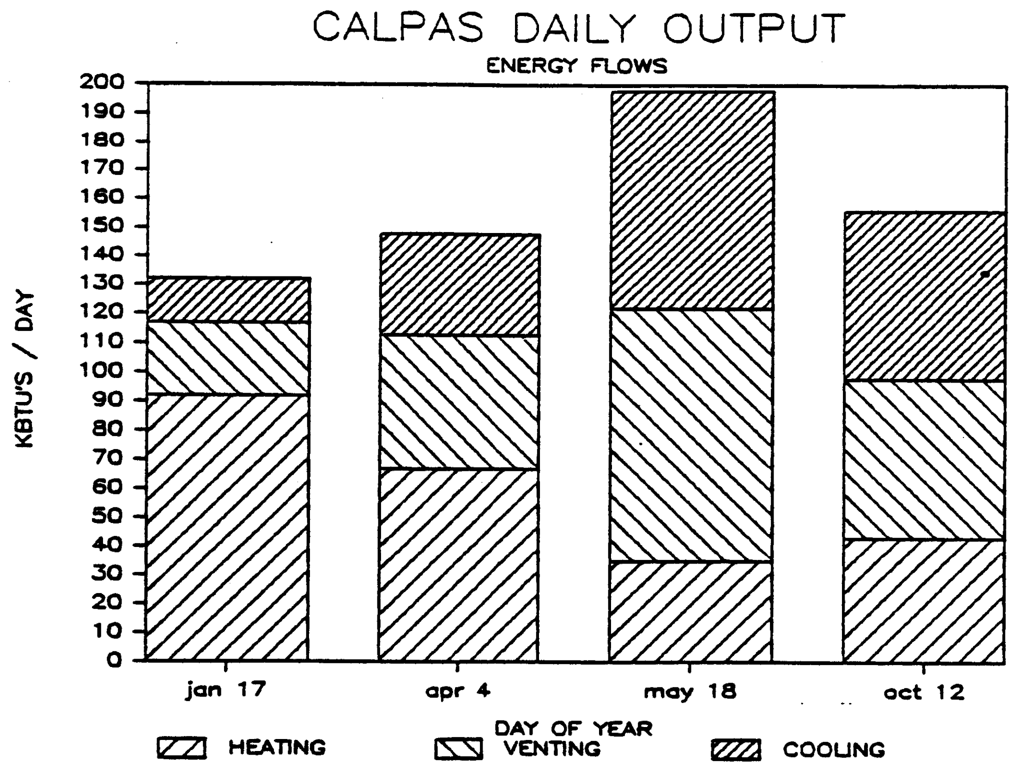
NAME OF FILE	HEATING KBTU'S	VENTING KBTU'S	COOLING KBTU'S
"lakin"	148	12763	741

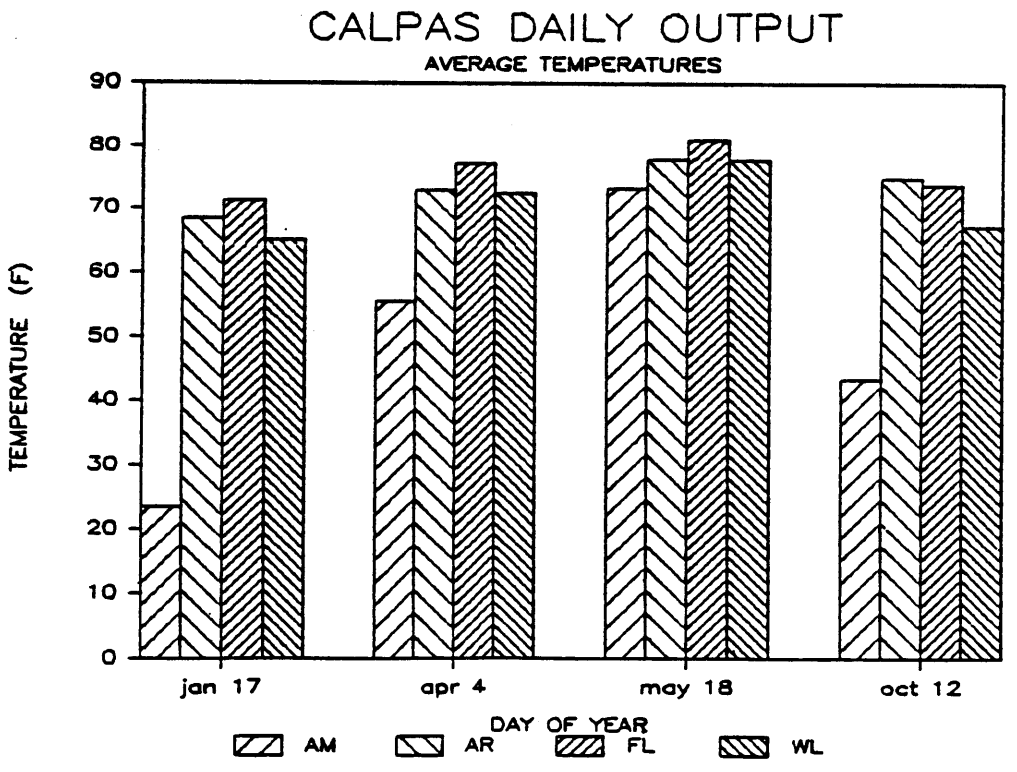
CALPAS YEARLY OUTPUT
HEATING & COOLING PEAK TEMPERATURES

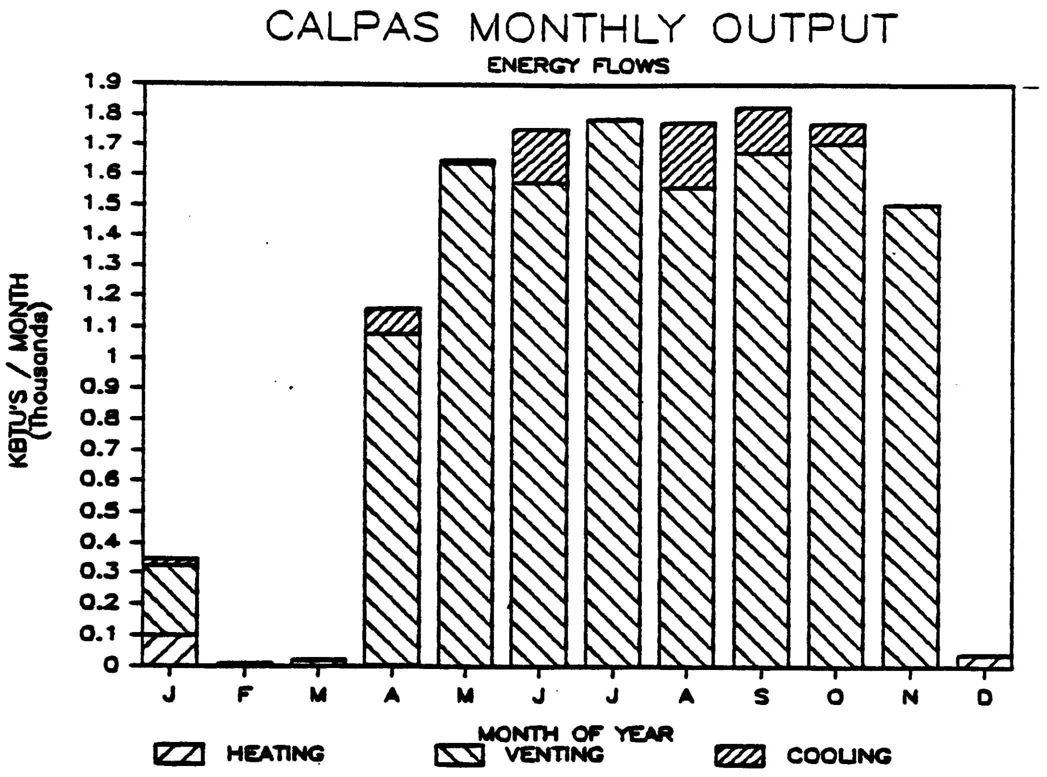
NAME OF FILE	HEATING PEAK	COOLING PEAK
"lakin"	5026	8191

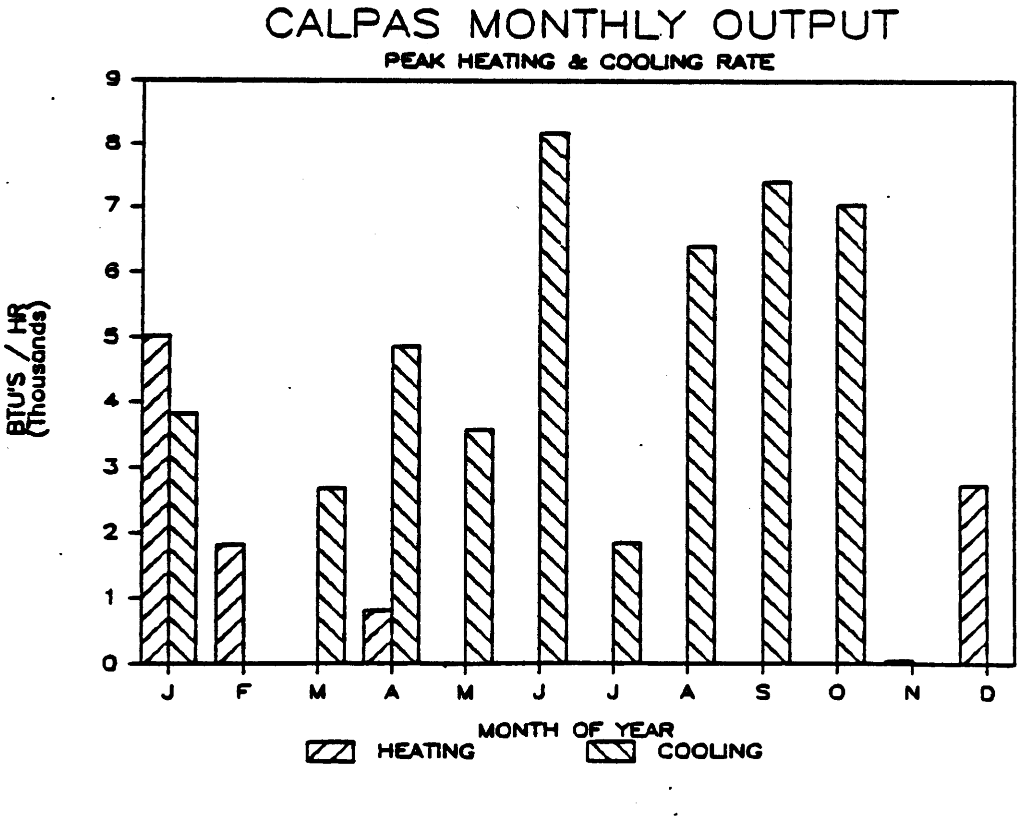

CALPAS HOURLY OUTPUT
SPACE & STORAGE TEMPERATURES
TEMPERATURE (F)
HOUR OF DAY
AIR
FLOOR
WALL
QUICK

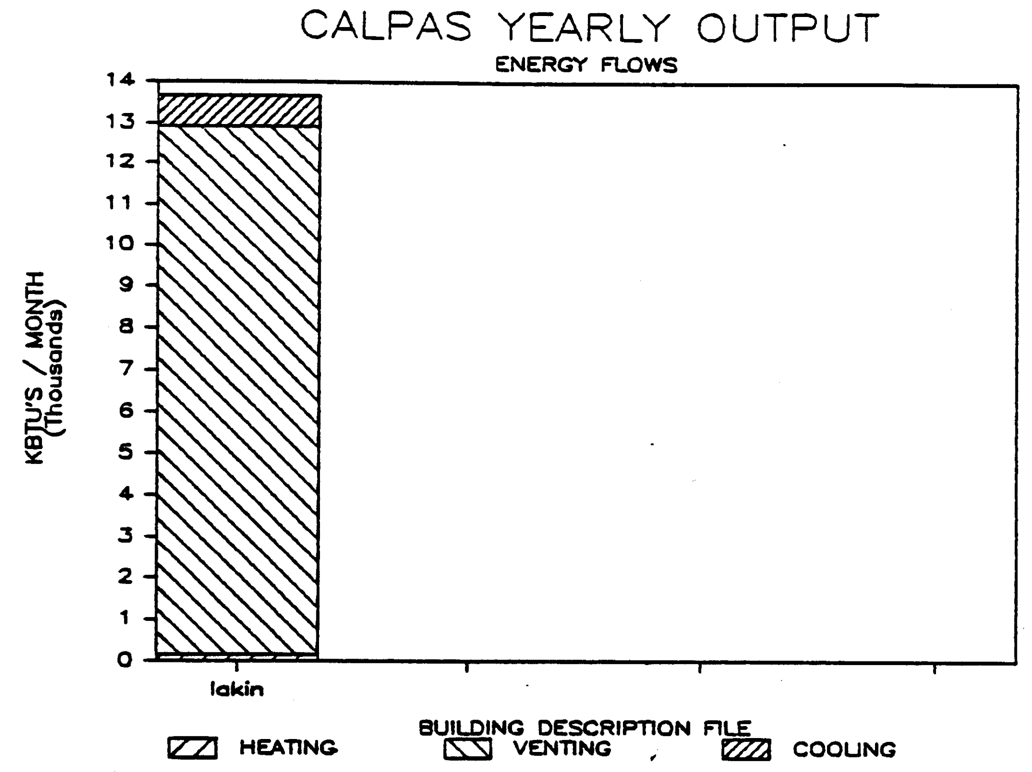

CALPAS HOURLY OUTPUT
ALL TEMPERATURES
TEMPERATURE (F)
HOUR OF DAY
AM
A1
A2
FL
WL
QK

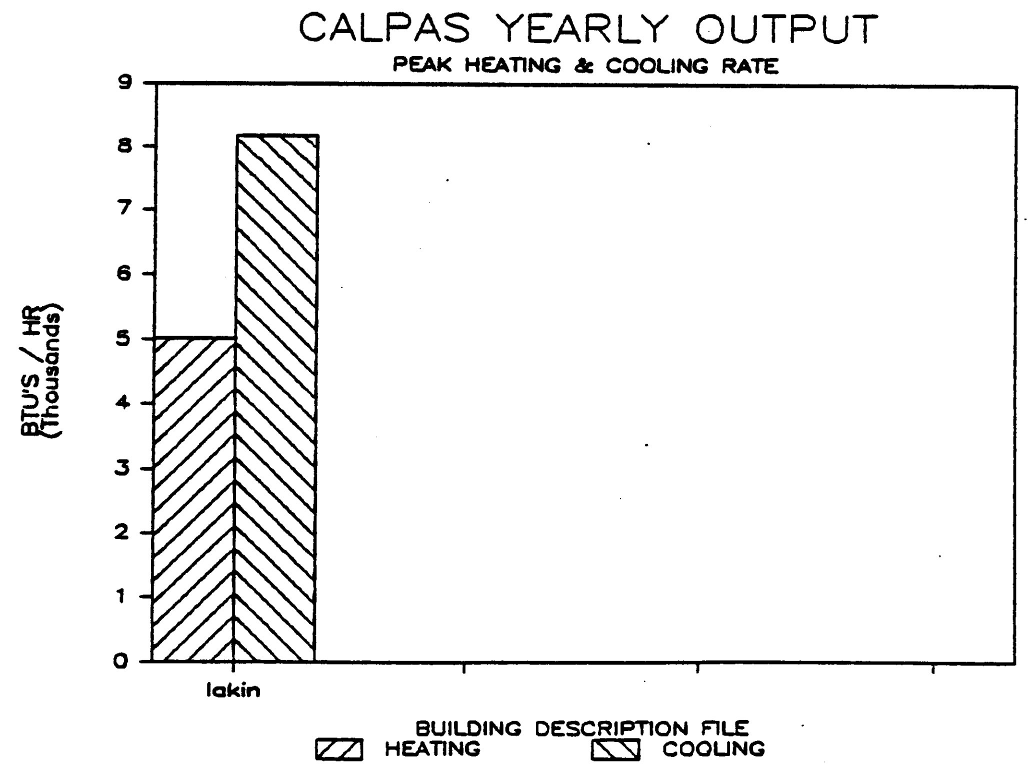

CALPAS DAILY OUTPUT
ENERGY FLOWS
KBTU'S / DAY
200
190
180
170
160
150
140
130
120
110
100
90
80
70
60
50
40
30
20
10
0
jan 17
apr 4
may 18
oct 12
DAY OF YEAR
HEATING
VENTING
COOLING


CALPAS DAILY OUTPUT
AVERAGE TEMPERATURES
TEMPERATURE (F)
90
80
70
60
50
40
30
20
10
0
jan 17
apr 4
may 18
oct 12
DAY OF YEAR
AM
AR
FL
WL


CALPAS MONTHLY OUTPUT
ENERGY FLOWS
KBTU'S / MONTH (Thousands)
1.9
1.8
1.7
1.6
1.5
1.4
1.3
1.2
1.1
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
J
F
M
A
M
J
J
A
S
O
N
D
MONTH OF YEAR
HEATING
VENTING
COOLING


CALPAS MONTHLY OUTPUT
PEAK HEATING & COOLING RATE
BTU'S / HR (Thousands)
9
8
7
6
5
4
3
2
1
0
J
F
M
A
M
J
J
A
S
O
N
D
MONTH OF YEAR
HEATING
COOLING


CALPAS YEARLY OUTPUT
ENERGY FLOWS
KBTU'S / MONTH (Thousands)
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
lakin
BUILDING DESCRIPTION FILE
HEATING
VENTING
COOLING


CALPAS YEARLY OUTPUT
PEAK HEATING & COOLING RATE
BTU'S / HR (Thousands)
9
8
7
6
5
4
3
2
1
0
lakin
BUILDING DESCRIPTION FILE
HEATING
COOLING