A MAN-MACHINE INTERFACE

FOR PC-CONTROLLED INJECTION MOULDING

BY

HARRY B. FUSSER

A thesis submitted to the Faculty of Graduate Studies

and Research in partial fulfillment of the

requirements for the degree of

Masters of Engineering

Department of Chemical Engineering

McGill University

Montreal

November 1991

Ich bin ein Teil von dieser Kraft, die stets das Böse will und doch

das Gute schafft...

Johann Wolfgang von Goethe (aus FAUST I)

ABSTRACT

In parallel to the development of a PC-control system for injection moulding, a workable user interface for man-machine communication was developed

The hardware chosen for enhanced process control includes an IBM PC/AT, PS/2 model 70. A real-time multi-task operating system, QNX 4.0 (Quantum Software Sys. Ltd.) was installed to run applications programmed in the language C (WATCOM Sys. Inc.).

The interface was tailored to satisfy the needs of casual users (operators) and professional users (researchers). A top-down structure was adopted, comprising menus, warnings, and directives.

The user interface includes three tasks to run concurrently, the main-task, the real-time machine-status display task, and the task to display the current barrel-heater temperatures. A mere expedient for a future runtime version, a dummy task can be started to display sensor locations and the machine in the four stages.

Four controller types, a digital PID and three discrete controllers, the total of cycles for the machine to run, and operating modes can be specified by scrolling sub-menus.

Activation of menu items is similar to the principles of commercial software. Protection against typing errors and optional detection of range violations is provided. A full-screen editor for configuration parameters or maximum and minimum values for each of the set-up parameters is provided.

All data processed on the loading, editing, and saving level, are character strings that will be converted to appropriate data types just before machine activation. The resulting magnitudes are optionally checked for range violations (safety barrier).

Inter-task communication is accomplished by sharing global memory segments between display tasks and the tasks in charge for data acquisition and control.

The interface tasks do not exert a large claim on system resources due to printing and rare screen upgrading in text mode, declaring and referencing global variables, and including user-defined header files into the code of modules.

Future improvements should focus on changing the default scheduling policy by QNX 4.0 and on the development of a graphical user interface on a separate computer.

Résumé

Un système de contrôle de procédé digital pour une machine de moulage par injection, faisant usage d'un IBM PC/AT, PS-2 model 70, ainsi qu'une interface homme-machine ont été développés à McGill (1991).

Un système d'opération multitâche QNX 4.0 (Quantum Software Sys. Ltd.) fut installé et rendit possible mise en action de plusieurs programmes écris en langage C (WATCOM-C, WATCOM Sys. Inc.).

L'interface est conçue de telle manière qu'elle puisse satisfaire les besoins des opérateurs et les besoins des chercheurs. La structure d'interface est composée d'une hiérarchie de menus, d'avertissements, et d'instructions.

L'interface éveille trois procédures (tâches) qui sont executées simultanément: la tâche paternelle, la tâche qui réalise l'indication du cycle présent et la phase à temps réel, et la tâche responsable d'indication des températures de fourreau. Une tâche provisoire fut conçue pour indiquer l'implantation des captures installées et les quatres phases de la presse à injecter.

En utilisant un méchanisme de séléction établis à l'aide de menus, il est possible de choisir entre quatres modes de contrôle, qui sont constitués par un régulateur de type PID et par trois autres types digitals.

L'activation des postes de menu est similaire à celle du logiciel commercial. Un mécanisme de protection contre des fautes dactylographiques et une examination des valeurs données, afin d'éviter qu'ils débordent les extrêmes des intervalles de limite, sont appliqués. Un

éditeur à plein écran peut être activé pour spécifier des données de configuration ou des valeurs de limite pour chaque valeur donnée.

Toutes les données procédées par l'interface sont des données ASCII enchaînées, jusqu'à ce qu'elles soient converties en types adéquats, avant que la presse à injecter soit mise en marche. Si l'opérateur le désire, les valeurs des données seront vérifiées pour savoir si les valeurs limites sont dépassées.

L'échange de données entre les tâches du système entier se base sur la déclaration des sections de mémoire (registre d'addresses de mémoire) globales, que les procédures d'affichage, de contrôle, et de collection de données partagent.

Les procédures d'interface ne consomment pas énormement de resources de système, parce que imprimer sur l'écran est effectué en mode d'édition de texte. De plus, comme les variables des programmes sont déclarées de telle manière qu'elles puissent être accessible globalement, des déclarations répétées sont évitées. Une autre raison, pour laquelle les procédures d'interface ne consomment pas énormement de ressources de système, est que des fichiers d'instruction début sont inclus dans les modules de code.

Des améliorations à venir appartiennent à une modification du principe d'allocation du processeur central entre les procédures et au développement d'un interface homme-machine effectué en mode graphique, ce qui devrait être réalisé par gestion des travaux au complexe d'ordinateurs.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my research supervisors, Professor M.R. Kamal and Professor W.I. Patterson, for their gratitude, support, patience, and encouragements throughout this project.

In addition, I wish to thank:

- The Department of Chemical Engineering of McGill University for teaching and research assistantship granted to me during the course of my studies.
- Mr. Furong Gao for his excellent introduction into the fields of real-time programming of applications for multi-tasking operating systems.
- Dr. Robert Di Raddo for his support to master disappointing situations regarding the academic procedures at McGill.
- Dr. Günter Lohfink and my father Mr. Heinz Fußer for their insistence that a withdrawal from the M.Eng program would be unwise.

Finally, I wish to thank all my colleagues in the department for having helped create a very stimulating social and professional environment in which to work.

CONTENTS

"C-Programming of a Man-Machine Interface

for PC-Controlled Injection Moulding"

Subject: Page:

Diskette 1 (MS-DOS format):

- source code (ASCII) of interface modules: '*.c'
- executable file and object modules library (QNX 4.0 machine code):
  - 'variable' - simulation task for control & data acquisition related tasks.
  - 'g40.lib' - object modules library for the module 'viewdat.c' (expedient for future runtime version). (cont.)

Diskette 2 (MS-DOS format):

- executable files and object-modules library 'if40.lib' (QNX 4.0 machine code):

(1) 'imm' - interface main task;

(2) 'statdip2' - real-time status display task;

(3) 'barreltemp' - display task for barrel heater temperatures;

(4) 'variable' - simulation task for control & data acquisition related tasks;

(5) 'if40.lib' - object-modules library;

LIST OF FIGURES

LIST OF FIGURES (cont.)

1. Introduction

For reasons associated with higher productivity, efficient utilisation of natural resources and uniform product quality, the use of microprocessor-based digital control continues to expand in the process industries. On all levels, the factors inhibiting computer applications for process control limits are constantly dwindling. Processing speed and capacity are increasing, and interfaces to the real world are continually performing more complex I/O (input/output) operations.

These capabilities suggests the use of a computer as an intelligent tool to manipulate and control the complex dynamic process of plastics injection-moulding . The configuration, however, of any digital, computer-based process-control system in this context is not obvious. This has led to extensive efforts aimed at clarifying how best to employ digital technology for improving the performance of the injection moulding process.

In addition to overall system configuration and component selection, it is necessary to develop a user interface supporting man-machine communication and ease of control system configuration. The objective of this work has been to design a user interface for application in conjunction with a PC-control system for injection moulding. The primary and secondary specifications that governed the user-interface design may be listed as follows.

(i) The interface must provide features required for three different operating modes (explained in section 2.1 (b): "user perspectives"). These modes are:

	LEVEL	SPECIFICATION
(a)	- level 1 / operator level	machine operations by default or preset conditions
(b)	- level 2 / professional user level	machine operations by explicit setup specification
(c)	- level 3 / research level	machine operations by explicit setup specifications, data acquisition, and control system configuration

(ii) The interface must exert a minimum claim on system resources (i.e. CPU time or memory).

(iii) The user interface must be synchronized and must communicate efficiently with the sub-systems related to digital control and data acquisition.

(iv) The interface program must feature an open structure that is uncomplicated to modify and to expand.

(v) The current machine-status and the barrel-heater temperatures must be displayed in real-time.

The secondary specifications, which were not considered to be as critical as the primary objectives, included the following:

(vi) Protection against typing errors and range violations must be provided.

(vii) Parameter input and user commands must be menu-driven (top-down structure).

(viii) The program structure must incorporate logical branching for the real-time display of sensor readings (to be developed in the future).

This thesis is divided into ten chapters. Chapter 2 supplies background information, which covers process-control technology for injection moulding and user perspectives, as well as issues affecting real-time multi-tasking operating systems. The chapter concludes with a description of components selected to establish the PC-control system for injection moulding at McGill (1991). In chapter 3, interface features are specified. Both chapters 2 and 3 deal with issues on the system level. The Chapters: 4,5,6, and 7 deal with, source code architecture, coding strategy, and debugging, respectively. Conclusions and recommendations concerning further work are given in chapter 8. A diagram depicting the interface menu levels and simplified flowcharts for the various menus are included in the appendix, chapter 10. The object-module listings (source code), the executable interface programs, and the library 'if40.lib' are stored on the included floppy diskettes. The C-code listings were converted from QNX 4.0-format to DOS/ASCII-format and are saved on diskette 1 in addition to the object modules library 'g40.lib' and the executable file 'viewdat' (QNX 4.0 machine code). Further executable files and the object modules libraries 'if40.lib' (QNX 4.0 machine code) are stored on diskette 2.

2. Background

2.1 Process control technology for injection moulding and user perspectives

(a) PROCESS CONTROL TECHNOLOGY

Currently, high performance injection moulding machines are equipped with a numerical control system (CNC). In fact, there is a trend towards the establishment of integrated systems for quality assurance (QAS), computer integrated manufacturing (CIM), production and maintenance scheduling (PMS), and distributed computer control (master/slave principle for supervisory control, Reiling, (1989), and Schwab, (1989)).

Due to its complexity, a sufficiently accurate dynamic model for the injection moulding process, which most control strategies would require, is not yet available. However, the statistical evaluation of process data often leads to valuable correlations between process parameters and quality features of moulded parts. As long as a substantial progress in modelling is missing, these correlations continue to build the backbone of (statistical) process control (SPC) systems in injection moulding (Schwab, (1989)).

It goes without saying that research in the field joins in following the industrial trends mentioned above, and to a certain extent focuses on subjects beyond the industrial scope. Industry has an interest in establishing (numerical) correlations between process parameters and quality features to produce fewer faulty mouldings. Research activities aim at improving physical models to better describe the nature of injection moulding for further advanced technologies. These models need to be verified. Then, powerful process data acquisition

features of numerical controls would tremendously help test process-control based on an improved model.

Often research engineers are to decide on whether to buy state-of-the-art machinery (complete unit) or to spend efforts on upgrading outdated machines (adding modular components) alternatively.

The first alternative is to purchase a CNC injection-moulding machine for studies of the injection-moulding process. Such a machine generally features a network of sensors, digital circuitry including ADC/DAC (analog to digital converters and vice versa), a single PLC (programmable logic controller) unit or a larger number, programming interface, data-bus interface, and a video system. These components are essentially required to perform QAS, PMS, and CIM found in the manufacturing environment of the modern factory. Most features of such a system are conceived to exchanging machine and process data (logging and loading) between the machine, host computers (data bases), and operator terminals. Aside from specifying machine settings and the logging modes, adjustments to the PLC program can be made via terminals. Nonetheless, the PLC unit must be re-programmed off-line to install a different program for control. However, it is usually not possible to employ other control strategies than SPC or digital PID control with PLCs. Finally, the usually significant contribution to cost of the above hardware components of CNC injection-moulding machines is not negligible, since research budgets tend to be limited.

The second alternative for research in injection moulding is to construct a modular process control and data-acquisition system that warrants greater flexibility regarding system configuration at lower cost. High-end personal computers equipped with modern data-

acquisition boards (short conversion time) are affordable and satisfy the requirements (refer to sections 2.2-2.4 for details) of process-data acquisition and digital control. Moreover, it is considered advantageous that control algorithms run on a PC can be changed easily, whereas programming flexibility of CNC-machines is more restricted. Where cost factors are important , and where it is intended to alter the system configuration frequently as well as to provide various operating modes, this approach is suitable for upgrading conventional injection-moulding machines found in research laboratories.

Individual control of the injection-moulding phases is a significant topic of research into process control in the Chemical Engineering at McGill (1991). Secondly, a variety of projects in a suite of on-going research requires experiments where the modified injection-moulding machine at McGill has to be operated in various configurations. Since these considerations as well as budget constraints coincide with the above arguments of the second alternative, a versatile PC-supported digital control-system was designed and constructed at McGill University (1991) bypassing analog controls of the injection-moulding machine.

(b) USER PERSPECTIVES

The search for optimum operating and configuration features of a PC-control system for an injection-moulding machine depends on three different perspectives: the manufacturing aspects, the professional user needs, and the research perspective. A suitable user interface must exhibit that these perspectives were taken into consideration. Their implications on the design of the user interface are summarized next.

All those users whose main interest is producing a varying number of moulded parts while accepting the default parameter-settings of the machine and control system are identified as "operators". Considering manufacturing on the operator level, the constraints on the interface design are as follows. The interface should "fail safe" when required command sequences are violated. System restart must be made possible without a need for invoking housekeeping routines on the operating-system shell after a shutdown. I/O requests should be menu-driven such that default settings or values are displayed (e.g. default file locations). If low-level menus suited to system configuration are entered accidentally, (automatic) entry and/or range-error protection, if applicable, has to be provided. Warnings and acoustic signals are required if system corruption or logical deadlock is at stake. Screen lay-out and colouring should be chosen in a way that memorizing menu sequences is supported.

Research work would involve a modification of the system's default configuration. This is the perspective of professional users. An alteration can pertain to a change of set-up parameters with the help of low-level menus (e.g. for the fast ADC sampling-rate at injection). On the research level, a professional user ("researcher") would modify both parameter settings for the process-control system and for machine set-up. In addition to the set of constraints originating from the manufacturing perspective, menus for set-up parameters have to be provided for professional users. Secondly, editing utilities have to be provided to satisfy researcher needs (for various sets of set-up parameters and range values). Finally, menus for configuration need to be added to those used by operators. Range-error and entry-error protection become a primary issue. Files holding minimum and maximum values for any given parameter have to be created. Selecting directories for storage of sampled-data files and locations for range data files should be menu-driven. If machine settings are to be tested that are beyond present experience, range checking must be suppressed. Identification of

appropriate minimum and maximum values for any given set of machine-parameter and control-parameter settings represents a research project in itself.

2.2 Real-time and multi-task computing for data acquisition and control

The minimum hardware and software requirements for real-time data acquisition and control are: a real-time clock, one or more digital to analog converters, an analog to digital converter or a larger number, and a real-time operating system. These must be considered integral parts of any process-control system incorporating a PC. The relationship of these components is depicted in Figure 1.

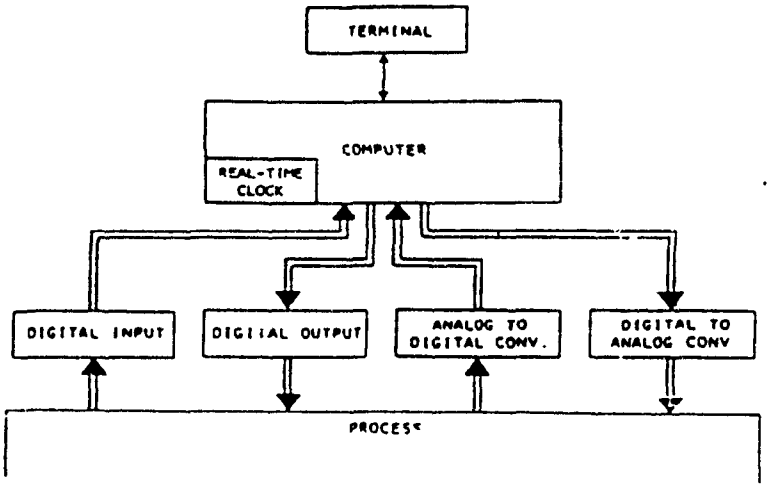


Figure 1: *Example of a Real Time System* (Mellichamp,(1983))

2.2.1 Working principles and conceptual overview of real-time multi-tasking operating systems

The following discussion gives a brief explanation of the characteristics of modern real-time multi-tasking operating systems.

2.2.1.1 Real-Time and multi-tasking features

System resources are limited (e.g. amount of RAM, CPU processing capacity, size of I/O buffers). Thus, real-time operations and multi-tasking are based on sophisticated mechanisms to obtain maximum performance.

A real-time multi-tasking operating system services interrupt requests for real-time I/O or for task switching. Interrupt requests are issued either by peripheral devices, by the system itself, by keyboard entry, or by user tasks (software interrupts, traps). Interrupt requests serve to suspend the processor's allocation to the current process. Servicing the process involved with the issue of the interrupt request is then accomplished virtually at once by the CPU. This ensures that unpredicted events in the outside world are acknowledged with almost no delay by the computer (real-time I/O).

Overall execution time, including a job mix of I/O bound and CPU bound processes, can be reduced when multi-tasking is applied (time advantage). Figure 2 illustrates the principal time advantage of multi-tasking compared to single-task operations. Aside from the acceleration of I/O for the obsolete devices such as teletypes and card-readers, a reduction of total I/O time due to multi-tasking can also be achieved when modern devices such as line printers and plotters are applied. No matter what devices are involved, during most of the time I/O is performed the CPU stays idle, waiting for the I/O data transfer (reading from and writing to I/O registers) to terminate. It then resumes activities to properly conclude the I/O process. The intermediate idleness of the CPU shrinks when multi-tasking is employed, since multiple I/O procedures can be initiated virtually at a time.

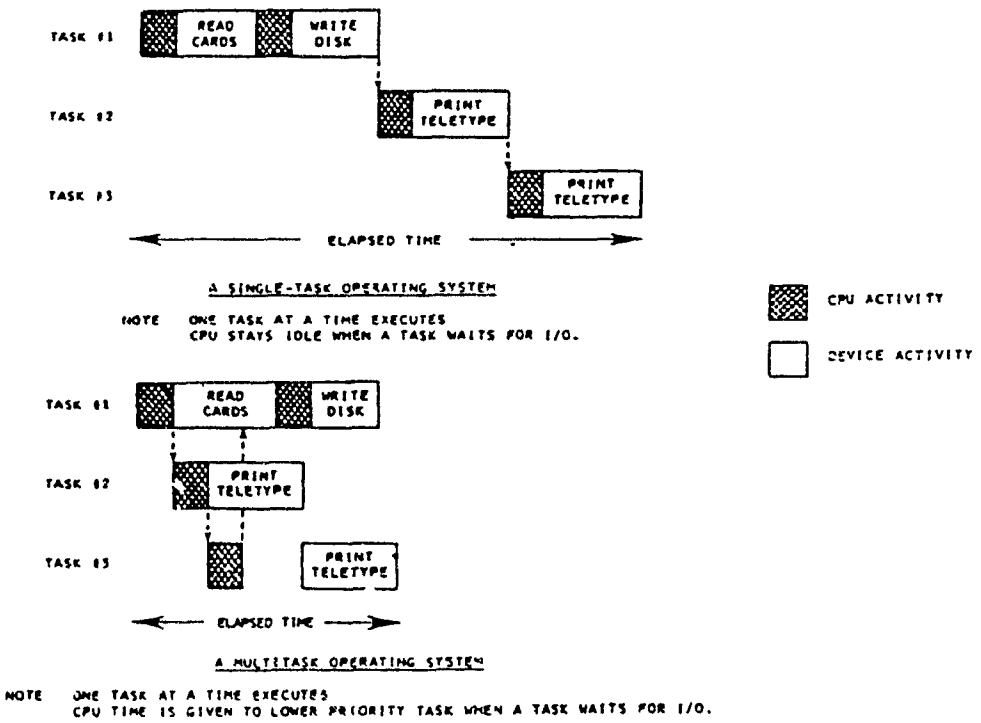


Figure 2: *I/O Sequence of a Multi-Tasking Operating system and a Single-Task Operating* (Mellichamp,(1983))

If no I/O operations are to be performed, it can still be desired that tasks run only during a defined time interval which is smaller than total execution time. Starting and halting tasks would then be the desired operating system feature leading to a hierarchy of tasks virtually run simultaneously.

Multi-tasking on single-processor machines is accomplished by time-slicing CPU-time. Allocation of the CPU to a process is allowed only for a fraction (slice) of its overall execution time. Interrupts are used for internal recognition of time-outs, and for switching CPU allocation. The software entity that performs slicing or implements the switching policy is referred to as a scheduler.

Several constituents of the operating system are involved with servicing interrupt requests for real-time I/O and task switching. These constituents are discussed next.

2.2.1.2 Operating system constituents

Some constituents of the operating system in low memory are used to store and retrieve information about process states (in a multi-tasking environment). These are referred to as system data-structures, and the processor queue. The entity known as kernel is the core of a layered system structure, where each layer is serviced dependent upon the priority assigned to it. The kernel is itself subdivided and comprises three sub-sections: the first-level interrupt-handler (FLIH), the dispatcher, and the processor queue mentioned above. The activities of these constituents (see sections 2.2.1.2 (c), (f), (g), and (h)) result in a change of the state of one or more processes present in the system (memory).

(a) PROCESS STATES

If the operating system is to handle switching the central-processor between processes, it has to keep track of the current state of all processes. A process state may be new, ready, active (or running), waiting (or blocked), or halted (slain). A process-state diagram is shown in Figure 3 below.

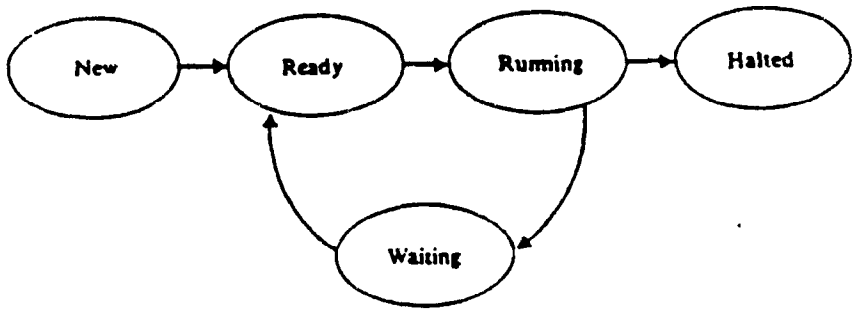


Figure 3: ***Process-State Diagram*** (Peterson,(1985))

(b) SYSTEM DATA STRUCTURES

The operating system applies a descriptive method, which is updating a complex data table in low memory, to keep informed about the overall state of process activities. Information about any process is saved in a process control-block (process descriptor). It contains the current status, other information, and specifications about the volatile environment of the process. This is the subset of the modifiable shared facilities of the system accessible to the process. The process descriptor of each process is linked into a process structure, which acts as a description to all processes within the system. The central table is a data structure to

serve as a means of access to all system structures. The central table has a pointer to each data structure and to other global information about the system.

(c) PROCESSOR QUEUE

All descriptors of runnable processes (in the ready state) are ordered by decreasing priority in a circular table known as a queue, so that the most eligible process (highest priority) is at its head. This processor queue includes descriptors to member processes that are ready to be allocated to the central processor. The linkage of the process structure and processor queue is in Figure 4.

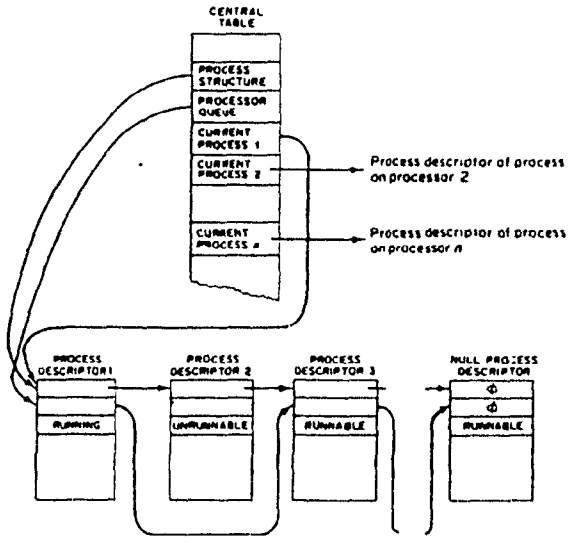


Figure 4: ***Process Structure and Processor Queue*** (Lister,(1988))

(d) SYSTEM LAYERS AND PRIORITY LEVELS

It is useful to visualize the structure of an operating system as the layered structure of concentric shells of an onion as shown in Figure 5.

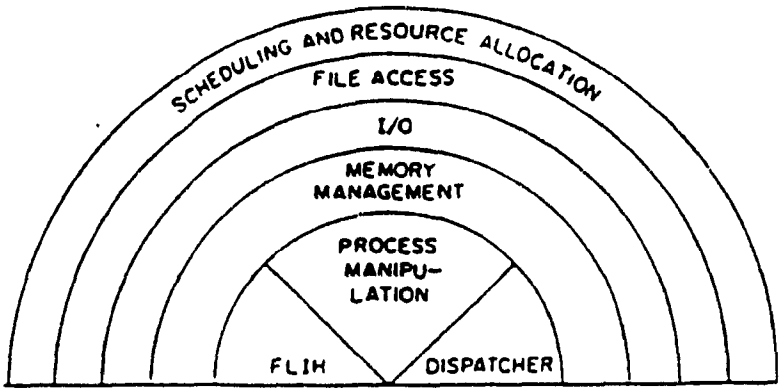


Figure 5: ***Layered Structure of a Real-Time and Multi-Tasking Operating System*** (Lister,(1988))

Figure 5 helps to show how time-sharing of the CPU depends on priority levels. Outside layers correspond to processes that are less time critical. They are given a low priority. In contrast, processes represented by inner layers run at higher priority, which means they are considered first when allocating the CPU anew.

(e) THE SYSTEM KERNEL

The most inner layer is referred to as the system's nucleus or kernel. The highest priority for CPU allocation is assigned to it. The kernel is itself divided into three modules (subroutines) namely: the first-level interrupt handler, the dispatcher (low-level scheduler), and the module in charge for process manipulation. The latter provides two procedures(routines) which implement the inter-process communication primitives "wait" and "signal" to be explained below (refer to section 2.2.1.3). These procedures are called via system calls (traps) in the processes concerned. The kernel is small, usually around 6Kbytes, since processing speed is urgent (negligible overhead). The real-time features depend on the kernel's organisation and its related performance. At the kernel level, it is required that two instructions are subsequently executed within a time interval of a fraction of a micro-second, whereas a time delay several orders of magnitude higher (up to milliseconds) is acceptable for the higher layers. Any process performed on a higher layer causes some activity of the kernel, particularly allocating the CPU to a process that requested to run (see (g), The dispatcher). It can thus be considered as a virtual CPU to any process or as a guard for the CPU.

(f) THE FIRST-LEVEL INTERRUPT HANDLER(FLIH)

The first-level (meaning lowest system level of highest priority) interrupt handler is the part of the operating system which is responsible for responding to signals both from the outside world (interrupts) and from within the computing system itself (system calls/traps/software interrupts). It should be distinguished from I/O device handlers and interrupt handlers included in application programs (higher level at lower priority).

The function of the FLIH is twofold:

(1) to determine the source of the interrupt;
(2) to initiate service of the interrupt;

The FLIH is always entered in supervisory mode, so that it has full access to the privileged instruction set (kernel procedures). Once invoked, the handler initiates saving of the program registers of the currently running process. Following the logic of a "skip chain", the handler next checks for the origin of the interrupt request. If a check results in a denial of an assumed source, the program counter of the handler "skips" to the next logical statement (if(source)then...) and the next source occurring in the skip chain is checked. This is continued until the source can be identified. Then the interrupt service-routine appropriate to the source is initiated. Scheduling mechanisms finally determine if the source process can run or has to wait. This is depicted in Figure 6.

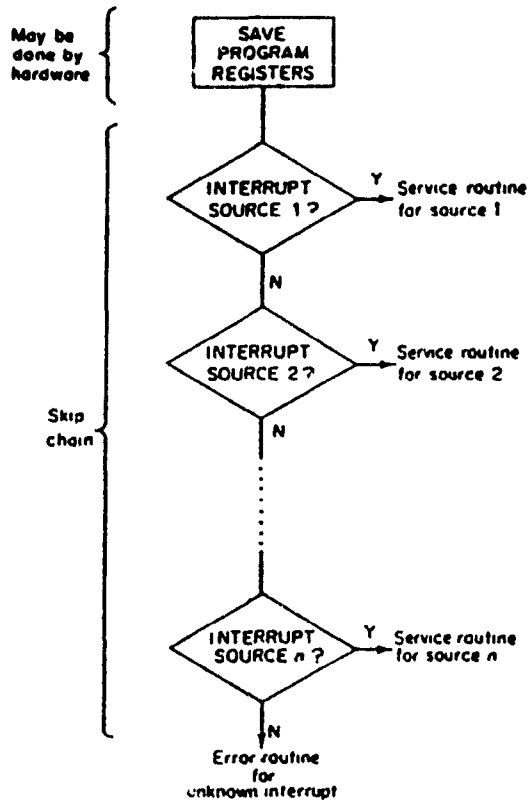


Figure 6: *Interrupt Identification by a Skip Chain* (Lister,(1988))

(g) THE DISPATCHER

The function of the dispatcher is to allocate the central processor among the various processes in the system. It is sometimes called as the low-level scheduler to differentiate it from the high-level scheduler (simply referred to as scheduler, see sections 2.2.1.6 (b) and (c) about scheduling) which is run at a lower priority. The dispatcher is entered whenever the current process cannot continue, or whenever the processor might be better employed in another process, i.e. after:

(1) an external interrupt, which usually changes the status of some process;
(2) a system call(trap), which suspends the current process;

An illustration about this is given in Figure 7.

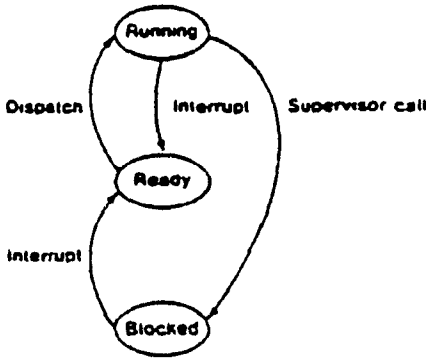


Figure 7: ***Invoking an Operation of the Dispatcher*** (Andrews,(1983))

In effect, the dispatcher is entered after all interrupts. The initial operation of the dispatcher is to check if the current process is still the most suitable to run. If not, it invokes saving the volatile environment - which is the subset of the modifiable shared

facilities of the system accessible to the process - of the current process in its process descriptor. Then the volatile environment of the most suitable process is retrieved from its process descriptor.

The operation ends by transferring control to the CPU to the newly selected process, at the location indicated by the restored program counter (Mellichamp, (1983), chapter 8; Lister, (1988); Andrews, (1983)). The most suitable process to run is identified by its priority. The assignment of priorities to processes is not the function of the dispatcher. This is the responsibility of the high-level scheduler described in section 2.2.1.5. As far as the dispatcher is concerned, the process priorities are given a priori.

(h) RELATIONSHIP between FLIH, DISPATCHER and PROCESSOR QUEUE

The basic synchronisation mechanism between the above constituents involves a "semaphore" (see 2.2.1.3 for more details). A semaphore is a non negative integer number which can either be incremented by one or decremented by one. It has the functionality of a flag that controls access to certain system routines and memory locations. Before the corps of a procedure can be executed, initial logical statements check for the current value of the semaphore associated with this procedure. After the occurrence of interrupts semaphores are employed to either awaken a process (signal) or to keep it suspended (wait).

The action taken by an interrupt routine to make a process runnable is twofold. Firstly, it must alter the status entry in the process descriptor, and secondly it must link the process descriptor into the processor queue at the position indicated by its priority. This can be done

by executing a "signal" operation on a semaphore. On the semaphore the process concerned has executed a "wait" operation. It is possible, that at a particular moment, a queue contains no processes. Rather than allowing a processor to loop within the dispatcher, it is convenient to introduce an extra process, called the null process, which has lowest priority and is always runnable. The null process may be nothing more than an idle loop, or it may perform some useful function such as executing processor test-programs. Its position is always the end of the processor queue (see Figure 4). With a knowledge of the processor queue and the FLIH, the operation of the dispatcher can now be summarised as follows:

(1) Is the current process on the processor still the first non-running process in the processor queue ? If so resume it. If not, then do (2) - (4).
(2) Save the volatile environment of the current process.
(3) Restore the volatile environment of the first non-running process in the processor queue.
(4) Resume running this process.

According to the above, it can be said that the overall real-time operation of an operating system is a result of the successful activation of the interrupt mechanism itself, the first-level interrupt handler, the interrupt service-routine, and the dispatcher, as indicated schematically in Figure 8.

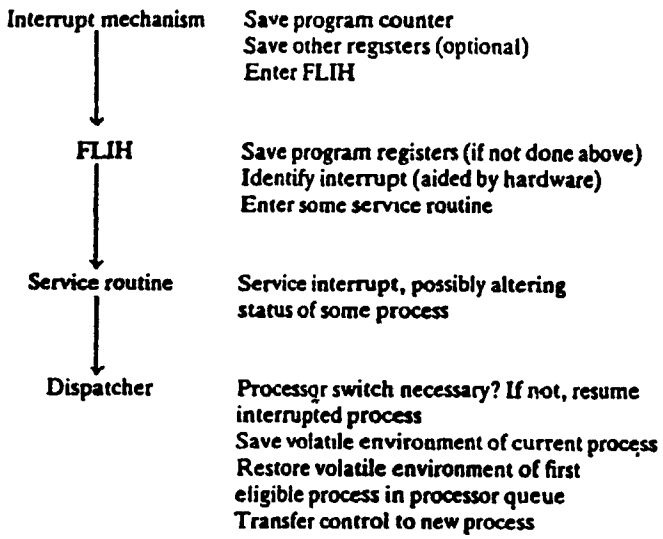


Figure 8: ***Relationship between FLIH, and Dispatcher*** (Lister,(1988))

2.2.1.3 Process manipulation by semaphores

A semaphore, s, can be understood as an unsigned integer on which either addition by one or subtraction by one is performed. It should be emphasized that the assignment:

$$\mathbf{s := s + 1;} \qquad (2.2.1.3)$$

is not the same as an increment operation on a semaphore. The difference arises from the fact that operations on semaphores are arranged to be indivisible. Indivisibility ensures that only one process at a time can execute an operation on the semaphore. It is important that a semaphore is properly initialized to a non-negative number. Binary semaphores can assume values of either one or zero. Counting semaphores can assume any non-negative integer, so that its initial value can be greater than one. By performing logical operations on semaphores, which are attached to processes, the process states (see section 2.2.1.2 (a): "process states") and the timing of instructions can be manipulated without creating huge and time consuming overhead. The mechanisms to implement indivisibility are to "lock" and to "unlock" the memory location where the value of the semaphore 's' is stored. Two methods are used: busy-waiting and interrupt inhibition. Both methods make use of the interrupt mechanism and differ only slightly (Lister, (1988)). These methods "lock" a semaphore before an operation by a process on it, and "unlock" it when the operation has terminated. Semaphores are used to quickly implement: blocking, unblocking, queuing, unqueuing, processor allocation, mutual exclusion, and task synchronisation.

(a) BLOCKING and UNBLOCKING

Blocking and unblocking are based on the kernel operations "wait" and "signal" that were mentioned above. In particular, the operations on semaphores to be implemented are:

wait(s) : **when** s > 0 **do:** decrement s;
signal(s) : increment s;

where 's' is the semaphore. The "wait" operation implies that processes are blocked when a semaphore has value 0 and freed when a "signal" operation increases its value to 1. One way to implement this, is to associate with each semaphore a semaphore queue. When a process performs an 'unsuccessful' "wait" operation (that is, it operates on a zero-value semaphore), it is added to the semaphore queue, and is made unrunnable. Conversely, when a "signal" operation is performed on a semaphore, some process can be taken off the semaphore queue (unless empty), and be made runnable again. Therefore, the semaphore is implemented with two components: an integer and a queue pointer. Logically, this concept can be described as follows:

wait(s) : **if** ≠ 0 **then** s:= s - 1
else add process to semaphore queue and make unrunnable;

signal(s) : **if** queue is empty **then** s:= s + 1
else remove some process from semaphore queue and make runnable;

The semaphore need not be incremented within "signal" if a process is to be freed, since the freed process would immediately have to decrement the semaphore again in completing its "wait" operation. Thus the "lock" and "unlock" operations are not substitutes for "wait" and "signal", since the latter are suitable for short delays whereas the lock/unlock operations consume considerably more time. Exclusive use of lock/unlock can lead to "thrashing" in systems with many tasks. Thrashing is the phenomenon where the supervisor spends much of its time managing the tasks leaving little time for their execution. However, the more processing capacity the central processor features the higher is the threshold number of tasks that can lead to noticeable thrashing.

(b) QUEUING and DEQUEUING

Semaphore queues that hold process descriptors indicating the "wait" stage of processes can be arranged in different forms. For most semaphores, a simple first-in first-out queue is adequate, since it ensures that all blocked processes are eventually freed. In some cases, it may be preferable to order the queue on some other basis.

(c) PROCESSOR ALLOCATION

The "wait" and "signal" operations may alter the status of a process, the former by making it unrunnable and the latter doing the opposite. An exit must therefore be made to the dispatcher for a decision on which process to run next.

In cases where no process status has changed (that is, a "wait" on a positive valued semaphore or a "signal" on a semaphore with an empty queue), the dispatcher will resume the current process, since it will still be the first non-running process in the processor queue.

The realisation of the "wait" and "signal" operations that comprises the mechanisms of: locking, blocking, queuing, the opposites of which, and processor allocation is illustrated in Figure 9.

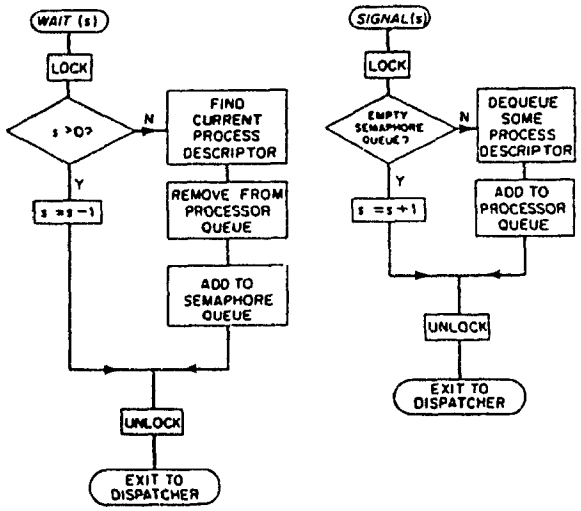


Figure 9: ***Implementation of "Wait" and "Signal"*** (Lister,(1988))

Mutual exclusion and task synchronisation are additional operations implemented by semaphores. The following paragraphs give a short description.

(d) MUTUAL EXCLUSION

Non-shareable resources (e.g. files, some data in memory or peripherals) can be protected from simultaneous access by several processes. This is accomplished by preventing the processes from concurrently executing the code which access the resources. These fragments of code are referred to as critical sections, and mutual exclusion in the execution of critical sections can be regarded as mutual exclusion in the use of resources. Exclusion can be achieved by the simple expedient of enclosing each critical section by "wait" and "signal" operations on a single semaphore whose initial value is 1. Thus, each critical section is programmed as ('mutex' is the semaphore name):

```
wait(mutex);
        critical section
signal(mutex);
```

(e) TASK SYNCHRONISATION

In a multi-tasking environment, tasks depend on intermediate results of other tasks, or system resources have to be allocated in a logical manner among multiple tasks. Then, task synchronisation becomes a primary issue. Generally, the tasks are said to be asynchronous. An example will clarify that execution of "wait" and "signal" on semaphores synchronizes processes in a straightforward manner. The simplest form of synchronisation is when a process, A, should not proceed beyond a point, L1, until some other process, B, has reached L2.

Examples of this situation arise whenever A requires information at point L1 which is provided by B when it reaches L2. The synchronisation can be programmed as follows:

```
Program for Process A        |    Program for Process B
          .                  |              .
          .                  |              .
L1:  wait(proceed);          |    L2: signal(proceed);
          .                  |              .
```

where 'proceed' is a semaphore with initial value 0. It is clear that A cannot proceed beyond L1 until B has executed the "signal" operation at L2.

2.2.1.4 The basics of dynamic memory allocation

Aside from memory segmentation and paging systems (Peterson, (1985), and Lister, (1988)), multiprogramming requires dynamic memory management. An important issue, thrashing common to multi-tasking is outlined below. This applies in particular if the size of working memory is limited.

Many programs exhibit behaviour known as operating in context. In any small time interval a program tends to operate within a particular logical module, drawing its instructions from a single procedure and its data from a single data area. The observation of this behaviour led to the postulation of the principle of locality (Denning, (1968)). It states that program references tend to be grouped into small localities of address space, and that these localities tend to change only intermittently. Based on this observation, the working set model of

program behaviour was developed. The working set model is an attempt to establish a framework of understanding the performance of paging systems in a multiprogramming environment. The competition for memory space between processes can in fact lead to behaviour which would not occur if each process ran separately. As the degree of multiprogramming rises, which is an increase of programs run concurrently, processor utilisation first also rises, since the dispatcher always has a greater chance of finding a process to run. However, when the degree of multiprogramming exceeds a certain level, then it is found that there is a marked increase in the paging traffic between main and secondary memories accompanied by a sudden decrease in processor utilisation. This phenomenon is called thrashing. The only conclusion to be drawn out of this is that each process requires a certain number of pages, called its working set, to be held in main memory before it can effectively use the central processor. If less than this number are present then the process is continually interrupted by page faults that contribute towards thrashing. To avoid thrashing only the working set of pages must be loaded into memory. By inspecting the process's recent history and a compliance with the principle of locality the working set, 'w', of a process at a time, 't', is identified to be:

$$w(t,h)=\{\text{page } i \mid \text{page } i \in N \ \&\ i \text{ appears in the last } h \text{ references}\} \qquad (2.2.1.4)$$

where 'h' is a parameter to indicate 'recentness' and 'i' is a non-negative integer (N comprises all integer numbers).

The larger is 'h', the further one looks into the past. Thrashing can be depicted graphically as shown in Figure 10.

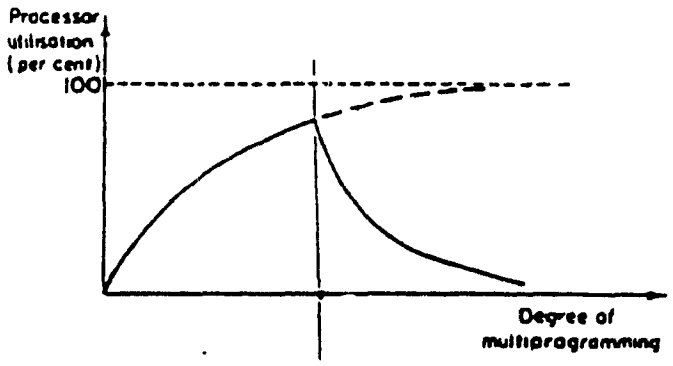


Figure 10: ***Thrashing example*** (Lister,(1988))

The above figure shows that if the working set theorem is negelected all tasks to be run concurrently will try to load as much code into memory as there is free memory available. The more tasks are added into the system the smaller is the number of pages of the added tasks that can be held in memory. At a threshold number of tasks, the most recently started tasks will have to swap pages from and to backing store (e.g. disk), since memory capacity is not sufficient to hold their entire working set. As a result, the CPU is blocked due to an disproportionate increase of page traffic.

As far as fetch and replacement policies are concerned, the significance of the working set lies in the following rule to which operating systems obey :

> Run the process only if its entire working set is in main memory, and never remove a page which is part of the working set of some process.

Once working memory is large enough to hold the entire instruction set of a process (real-time system), dynamic memory allocation becomes oblivious. Advanced systems perform memory compaction to reduce fragmentation (Mellichamp, (1983)).

2.2.1.5 The I/O subsystem

In order to reduce overhead due to copying of the same set of instructions, a device independent I/O mechanism is supported by the operating system. It means that programs do not operate on physical devices but on virtual devices known as streams.

Device characteristics are encoded in device descriptors, where one descriptor exists for each device in the system. These encoded device features are referred to by device handlers, the routines which provide instructions to handle the devices. There are separate device handlers for each device but they show great similarities. Differences of operations are derived solely from parametric information (read from the device descriptors !). Device handlers can make use of shareable programs.

The overall I/O process can be summarised as the joint effort of a requesting process, the actual I/O procedure, the device handler, and an interrupt routine.

A typical I/O request from a process will be a call to the operating system (trap) of the general form:

DO-I/O (stream,mode,amount,destination,semaphore)

where:

DO-I/O	is the name of a system I/O procedure;
stream	is the number of the stream on which I/O is to take place;

mode indicates the type of operation (data transfer or rewind) and the character code to be used;
amount is the amount of data to be transferred if any;
destination is the location into which (or from which) the transfer, if any, is to occur;
semaphore is the address of a semaphore called 'request serviced' which is to be signalled when I/O is complete;

The I/O procedure is re-entrant, so that it may be used by several processes at once. Its function is to map the stream number to the appropriate physical device (keyboard, card reader, floppy drive, printer, terminal, disk), to check the consistency of the parameters supplied to it, and to initiate service of the request.

When the checks have been completed the I/O procedure assembles the parameters of the request into an I/O request block (IORB) which it adds to a device request queue of similar blocks which represent other requests for use of the same device. These other requests may come from the same process, or, in the case of a shared device such as a disk, from other processes. The device request queue is attached to the descriptor of the device concerned and is serviced by the device handler (Lister, (1988)).

The device descriptors include a pointer to the process descriptors of the current process using the device.

The device handler is responsible for servicing the request on a device request queue and for notifying the originating process when the service has been completed. It operates in a continuous cycle during which it removes an IORB from the request queue, initiates the corresponding I/O operation, waits for the operation to be completed, and notifies the

originating process. An input operation cycle may be listed as follows:

```
repeat  indefinitely
begin   wait(request pending);
        pick an IORB from request queue;
        extract details of request;
        initiate I/O operation;
        wait(operation complete);
if error then plant error information;
        translate character(s) if necessary;
        transfer data to destination;
        signal(request serviced);
        delete IORB;
end;
```

'Request pending' is a semaphore which is contained in the device descriptor and is signalled by the I/O procedure each time it places an IORB on the request queue. 'Operation complete' is another semaphore that is signalled by the interrupt routine after an interrupt is generated for this device. The semaphore 'request serviced' is passed to the device handler as a component of the IORB. It is supplied by the process requesting I/O as a parameter of the I/O procedure.

The synchronisation of the I/O operation makes use of the mutual exclusion, and block and unblock features. This mechanism is shown graphically in Figure 11.

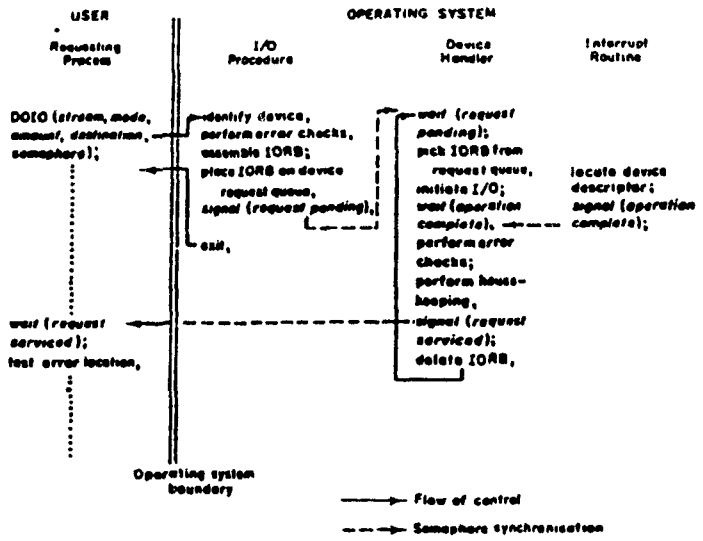


Figure 11: *Sketch of the I/O System Components* (Lister,(1988))

It should be mentioned that buffered I/O techniques are widely implemented to support multi-tasking on input and output. There are subtle differences if the devices are considered shareable (e.g. disk) or non-shareable (e.g. line printer). The latter involves a spooler that manages storing of files on a bulk storage devices before submission to the device is performed (Peterson, (1985)).

2.2.1.6 Resource allocation and scheduling

In an environment in which resources are limited a 'grab it when you need it' method of acquisition is rarely feasible to satisfy the concurrent demands of all processes in the system. Scheduling techniques are devised to share a limited set of resources among a number of competing processes. The objectives of these techniques are:

(1) mutually exclude processes from unshareable resources;
(2) prevent deadlock (see below);
(3) ensure a high level of resource utilisation (e.g. prevent thrashing);
(4) allow all processes an opportunity of acquiring the resources they need within a 'reasonable' time;

Deadlock and scheduling are described next.

(a) THE DEADLOCK PROBLEM:

In concurrent programming, a process sometimes must wait until a particular event occurs. If the event takes place and the waiting process is awakened, no problem arises. But if the event never occurs, the process will be blocked forever. A process is deadlocked when it is waiting for an event that can never occur. In a simple example such a situation arises when two processes compete for allocating the same resources. Say a process, P1, has a resource, R1, allocated while another resource, R2, is allocated to a process, P2. For further progress P1 requests R2 keeping R1 allocated. Process P2 still uses resource R2 and requests resource R1 which is currently allocated to process P1. It is obvious that both processes "wait" for

appropriate resources to be released. The deadlock arises, since for both processes the release of the resource kept by the competing process is a necessary condition for continuation. An allocation and request structure, in the form of a state graph, illustrates a deadlock if it shows a cyclic pattern that resembles a traffic deadlock involving a square of perpendicular one-way roads. Figure 12, shown below, will clarify process deadlock.

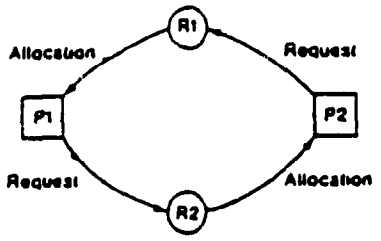


Figure 12: ***The Deadlock Problem in a Multi-Tasking Environment*** (Peterson,(1985))

The problem of deadlock may be solved, depending on the operating system at hand, by adopting *one* of the following strategies:

(1) **Prevent deadlock** by ensuring at all times that at least one of the four conditions above does not hold;
(2) **Detect deadlock** when it occurs and then try to **recover**;
(3) **Avoid deadlock** by suitable anticipatory action;

Excellent explanations can be found in literature (Lister, (1988), and Peterson, (1985)).

Scheduling is supported by case sensitive mechanisms which are ruled by various scheduling policies. The mechanisms and policies are summarised in the following paragraphs. Software entities referred to as long-term scheduler, medium-term, or short-term scheduler are integral parts of a complex system for scheduling.

(b) SCHEDULING MECHANISMS

The operating system's schedulers are in charge for introducing processes to the system, and withdrawing them according to: priorities, outside world events, and system-housekeeping requirements. These subjects are intimately related to resource allocation. In fact, decisions on scheduling and decisions on resource allocation are so closely linked that responsibility for both is often delegated to a single system process. This is particularly true for the scheduler mentioned earlier. The scheduler is more precisely referred to as high-level scheduler according to its priority or a long-term scheduler when processing time is emphasized.

In a scheduling context, a process or task is a program in execution. As the program executes, the process changes state as a result of its inherent logic. The state of a process is defined by its current activity. Scheduling decisions force a task to switch from one state to another.

A process is said to endure a transition between two succeeding states. A process in one of the four states: running, ready, inactive, or suspended can be forced to resume one of the states that is not the current state, which can graphically be depicted in a state diagram, Figure 13.

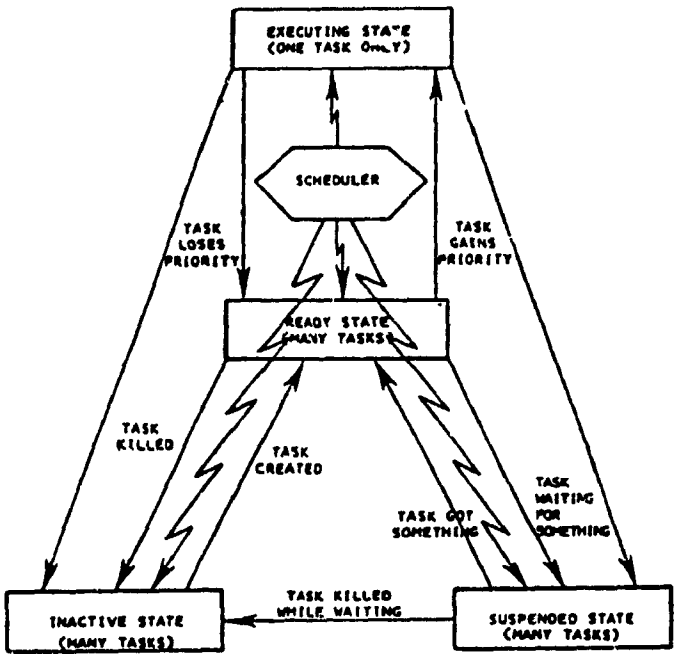


Figure 13: ***Task States, Transition, and Scheduler Signals*** (Mellichamp,(1983))

The scheduler makes use of the signalling mechanism to initiate a state transition along a transition path. Signalling a task currently in the ready state, for instance, can subsequently set it inactive (shown as a zig-zag path in Figure 13).

The kernel task referred to as dispatcher is in charge of actually allocating the CPU to processes requesting execution. With regard to its priority it is called low-level scheduler or short-term scheduler when processing time is emphasized. If the short-time scheduler is not a kernel task, there is a subtle distinction between the dispatcher (fastest scheduling operation)

and the short-term scheduler run at a high priority but slower than the dispatcher.

The primary distinction between the long-term scheduler and short-term scheduler is the frequency of execution. The short-term scheduler must select a new process for the CPU quite often, sometimes every 10 milliseconds. It must be very fast. As a rule of thumb for modern computers, it takes one millisecond to decide to execute a process for 10 milliseconds. Then 1/(10+1) ≈9% of the CPU time is being wasted simply for scheduling the work. CPU consumption for operating system related tasks is generally referred to as latency or latency time and is not negligible when high speed applications tasks are to be run.

The long-term scheduler, on the other hand, executes much less frequently. It may be seconds or even minutes between the arrival of new jobs in the system. The long-term scheduler controls the degree of multiprogramming, which is the number of processes in main memory. Except from having more time to decide on selecting jobs for execution, it is also important that the long-term scheduler selects a good job-mix of I/O-bound and CPU-bound jobs to keep CPU utilization balanced by preventing idleness or overflow.

The CPU allocation to I/O-bound jobs and CPU-bound jobs unfolds periodically. A new process is added to the processor queue (ready queue) after a decision by the long-term scheduler was made. It is picked up by the short-term scheduler once it has reached the position of the first non-running process in the ready queue. After being run for a time slice of CPU time, scheduling policies as implemented in the system force the process's pre-emption, since another process has acquired the position of the first non-runnable job in the system. If not yet completed, CPU-bound processes are generally placed back into the processor queue according to the scheduling policy. After some waiting time has elapsed it will be picked up by

the dispatcher again to run another time slice or to run to completion. This is illustrated by the Figure 14 below.

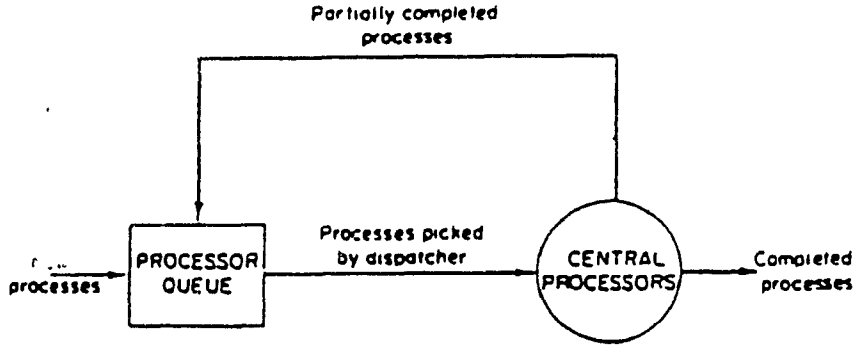


Figure 14: ***Scheduling Model for a Processor-Bound Process*** (Lister,(1988))

Since the I/O subsystem maintains a device request queue for I/O request blocks, which is linked to the processor queue, I/O-bound jobs are first placed into the I/O request queue (waiting queue) after having been initialized (assembling of I/O request block ...). Partially completed, pre-empted, I/O processes are placed back into the processor queue according to the scheduling policy by the long-term scheduler to await the dispatcher's attention. This is illustrated in Figure 15 underneath.

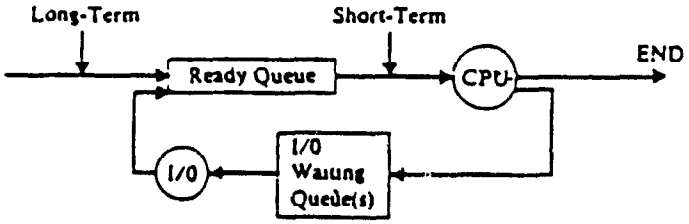


Figure 15: ***Scheduling Model for an I/O-Bound Process*** (Peterson,(1985))

The general scheduling model of an operating system includes a semaphore queue. Generally, CPU-bound processes, interrupted by a peripheral device or in the case of a foreign resource request, can be blocked and discarded into a semaphore queue. This would occur according to the deadlock prevention scheme and scheduling policy. Figure 16 illustrates the general scheduling model.

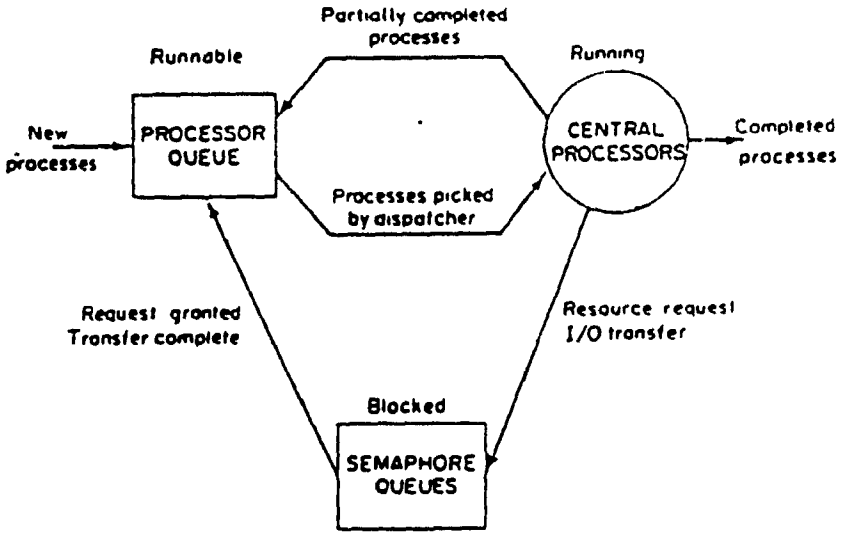


Figure 16: ***General Scheduling Model involving a Semaphore Queue*** (Lister,(1988))

(c) SCHEDULING POLICIES

Scheduling policies define decision making on topics which affect the positioning of process descriptors in ready queues, the order of process descriptors in semaphore queues and the updating frequency of the queues. A classification scheme for processes to be run must ensure that no ambiguous situations occur whenever the schedulers are invoked. In any event,

whatever scheduling issue unfolds to the schedulers, a decision about what process to make runnable next has to follow virtually immediately. Scheduling based on measurable quantities that are common to all processes always leads to a clear decision by comparison. Efficient scheduling policies undertaken by operating systems thus compare CPU utilization of individual processes. The methods of comparison may differ according to the scheduling policy employed.

A closer examination of processes reveals that a process is a program in execution consisting of an alternating sequence of CPU and I/O bursts, beginning and ending with a CPU burst . Although they vary greatly from process to process and computer to computer, they tend to have an exponential distribution of number versus burst duration. There are a very large number of very short CPU bursts and a small number of very long ones. An I/O-bound program would typically have many short CPU bursts. A CPU-bound program might have a few very long CPU bursts. This distribution can be quite important in selecting an appropriate CPU scheduling algorithm.

It may briefly be indicated that Gantt charts illustrate very well the resulting scheduling pattern for a given job arrival-sequence and individual CPU burst-durations, which determine a scheduling algorithm. A measure, known as average turnaround time, provides a means to evaluate an algorithm's performance. It is given by:

$$t = \frac{1}{n} \sum_{i=1}^{n} (j_{i,a} - j_{i,t}) \qquad (2.2.1.6)$$

where 't' equals the average turnaround time, 'n' is the number of jobs, '$j_{i,a}$' indicates the job

arrival time, and '$j_{i,t}$' indicates its termination time. A simplistic evaluation scheme states that an algorithm is better suited to a mix of tasks the lower 't' turns out to be (minimization of total execution time). This is valid if a minimum turnaround time is sought for quick overall task execution, whereas the sequence of CPU time quantums allocated to the tasks is irrelevant.

Pre-emptive algorithms are developed such that a newly arrived process in the processor queue will pre-empt the current process's CPU allocation if its priority runs higher. A non-pre-emptive CPU scheduling algorithm will simply put the new process at the head of the queue if its priority runs higher. In a time-sharing system this can be particularly troublesome. It is very important that each user or each task gets some share of the CPU at regular intervals.

The round-robin scheduling algorithm is designed especially for time-sharing systems. The Gantt-chart for which, Figure 17, is given as an example below.

Job	Burst Time
1	24
2	3
3	3

If we use a time quantum of 4, the resulting round-robin schedule is:

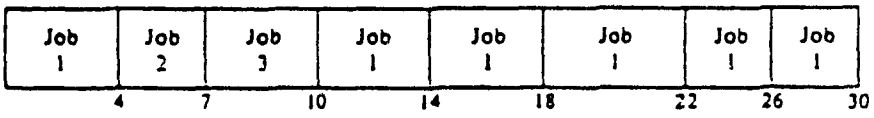


Figure 17: ***Gantt Chart for round-robin Scheduling*** (Peterson,(1985))

A small unit of time, called a time quantum or time slice, is defined. A time quantum is generally from 10 to 100 milliseconds. The ready queue is treated as a circular queue.

The dispatcher goes around the ready queue, allocating the CPU to each process for a

time interval up to a quantum in length. To implement round-robin scheduling, the ready queue is kept as first-in first-out queue, having the currently running job as the first entry in the queue. The dispatcher picks the first job from the ready queue, sets the timer to interrupt after one quantum, and dispatches the process. One of two things will then happen. The process may have a CPU burst less than the time quantum. In this case, the process itself releases the CPU voluntarily, by issuing an I/O request or terminating. Otherwise, if the CPU burst of the currently running process is larger than the time quantum, the interval will expire and cause an interrupt to the operating system. The registers for the interrupted process are saved in its process control block, and the process is put at the tail of the ready queue.

A trade-off between a decreasing time quantum and the increase of required context switching sets a lower limit to quantum shrinkage. Figure 18 below shows the increase of context switching by reducing the quantum.

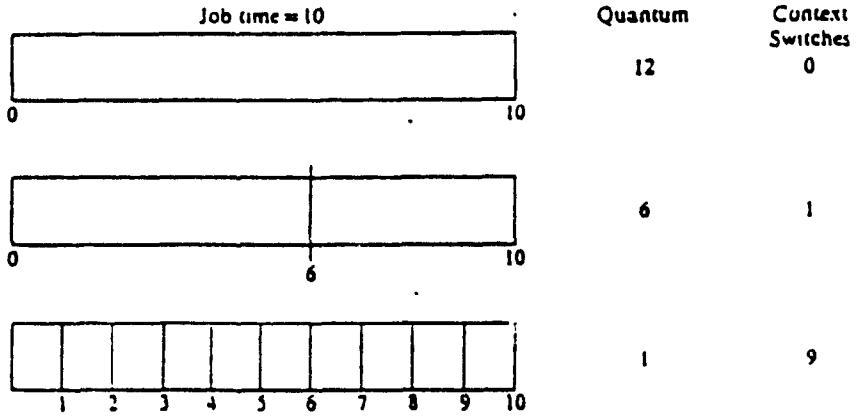


Figure 18: *Quantum Shrinkage and Increase of Task Switching Overhead* (Peterson,(1985))

The average turnaround time does sensibly drop when too small a time slice is selected. On the other hand, if the time quantum is too large, round-robin degenerates to a first come, first served processor queue. A suggested rule-of-thumb is that 80% of the CPU bursts should be shorter than the time quantum.

2.2.2 Multi-task programming principles

Multi-task programming is commonly accomplished by using task creation primitives and task communication primitives. Details concerning multi-task programming and grounds to apply it for PC supported digital control will be addressed thereafter.

(a) TASK-CREATION PRIMITIVES

Several logical constructs for establishing inter-task relationships are required. Each of those constructs can be illustrated by a precedence graph which shows the historical number and hierarchy of tasks preceding the ending of the graph. The "fork" primitive, as shown in Figure 19, allows a task to branch out into two concurrent tasks.

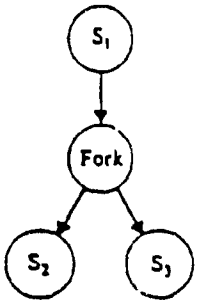


Figure 19: *Precedence Graph for the "FORK"-Construct* (Peterson,(1985))

The reverse logic is accomplished by the "join" construct which causes two preceding tasks to terminate while another task is started. A "concurrent statement" causes one task to terminate while several subsequent tasks are being started that run virtually simultaneously. This can be derived from the Figure 20.

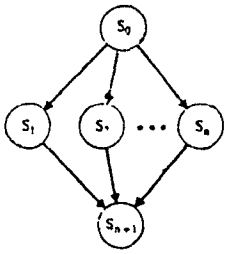


Figure 20: ***Precedence Graph for the Concurrent Statement*** (Peterson,(1985))

It should be mentioned that a concurrent statement can always be simulated by considering only fork/join constructs.

These constructs, supported by the operating system, are symbolised by a set of specific instructions (system calls) found in a programming language. It allows a programmer to establish a program hierarchy. The subsequent sketch, Figure 21, illustrates a program hierarchy. A, B, C ... J are tasks currently in the system.

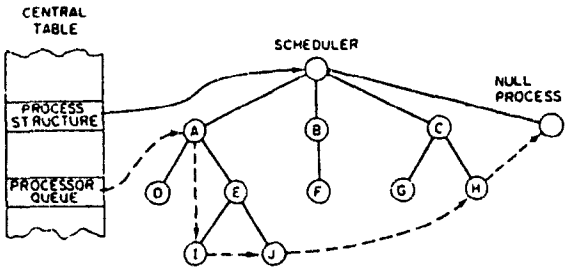


Figure 21: ***Program Hierarchy*** (Lister,(1988))

The scheduler (refer to section above) is responsible for initiating new processes, and acts as the parent of all processes (tasks) introduced into the system. It is responsible for the welfare of its offspring in that it governs policies of resource allocation and influences the order in which processes are selected by the dispatcher. This parental role is not restricted to the scheduler if multi-task programming is supported. Other tasks are given capabilities to:

(1) create subprocesses;
(2) allocate to their subprocesses a subset of their own resources (the resources are returned when the subprocess terminates);
(3) determine the relative priority of their subprocesses;

The creation of a hierarchy of processes is initiated by the logical order of the system calls mentioned above.

For these system functions, additional programming language instructions have to be provided. Multi-task programming features are strongly determined by the operating system which is supposed to enforce and maintain the program hierarchy.

(b) INTER-TASK COMMUNICATION PRIMITIVES

Other groups of system calls supported by the operating system handle inter-task communication. The very first technique available is sharing memory segments. The segment tables private to two processes include into particular segment descriptors the same base address of the memory space to be shared.

This is illustrated in Figure 22 below.

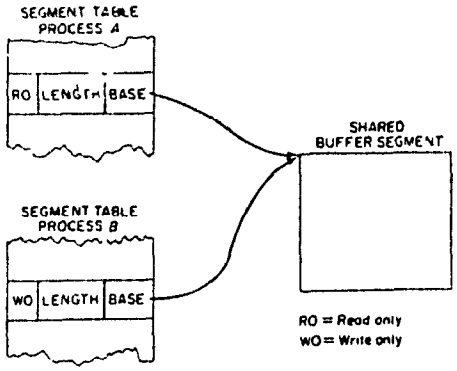


Figure 22: ***Segment Sharing with different Access Privileges*** (Lister,(1988))

Moreover, the access rights, either "read only" (RO) or "write only" (WO), are placed into the segment descriptors. In programming languages this feature is managed by pointer arithmetic and requires deep knowledge about a computer's architecture.

Direct communication is accomplished by referring to task names for sending and receiving messages. The "send" and "receive" primitives may be defined as follows:

send (P, message) : Send a message to process P.
receive (Q, message): Receive a message from process Q.

This link is bidirectional (symmetric) and affects exactly two communicating processes. It functions automatically, meaning the operating system establishes the link.

Asymmetric communication allows a process to receive messages from any process by the following scheme:

send (P, message) : Send a message to process P.
receive (id, message): Receive a message from any process; 'id' is set to the name of the sending process;

With indirect communication, the messages are sent to and received from mailboxes. The communication primitives in this case have a syntax as follows:

send (A, message) : Send a message to mailbox A;
receive (A, message): Receive a message from mailbox A;

In this scheme, a communication link is established only if the mailbox is being shared. Any link may be associated with more than two processes. Furthermore, between each pair of processes there may be a number of different links, each corresponding to one mailbox. The link may be either unidirectional or bidirectional.

A link has some capacity that determines the number of messages that can temporarily reside in it. This property can be viewed as a queue of messages attached to the link. A zero capacity queue (unbuffered sending) is established when waiting of messages is not tolerated (send blocking). In this case the sender must "wait" until the recipient receives the message. The two processes must be synchronized

for a message transfer (a rendezvous is established). The communication primitives may look like:

Process P executes:	Process Q executes:
send (Q, message);	**receive** (P,message);
receive (Q, message);	**send** (P,"acknowledgement");

Once the queue has finite length, it is said to have bounded capacity, several messages can reside in it. If the queue is not full when a new message is sent, it is placed in the queue (either by copying the message or by keeping a pointer to it), and the sender can continue execution without waiting. If the link is full, the sender must be delayed until space is available in the queue. A queue is of unbounded capacity if there is virtually infinite space such that any number of messages can wait in it. The sender is never delayed in this case.

The messages sent or received can be of fixed size, of variable size, and can be typed. The latter is applicable to mailboxes declared in strongly-typed program languages.

(c) BASICS of MULTI-TASK PROGRAMMING for PROCESS CONTROL

Process-control applications require performance of several tasks. The basic control scheme incorporates the collection of process data through ADC channels. The settings for final control elements according to a control algorithm are computed subsequently. Output data are finally sent to one or more DAC channels. The operations above (signal processing sequence in a feedback loop for control) have to be repeated periodically to adjust data

acquisition and controller output (computer) to the most recent state of the dynamic process to be controlled. The state parameters of dynamic processes vary as a function of time. Timing has to be accurate and is best accomplished by an interrupt handler to service real-time clock interrupts. The handler signals a counter to decrement its value (software timer). The same set of tasks is called repeatedly and has to stay resident in memory to limit overhead that would be incurred by swapping of code segments with bulk-storage devices.

Multi-tasking is highly recommended, since it does not require an exact knowledge of the sequence and time of events in advance. It may even be the only feasible approach for process control on a PC if processor idleness during I/O cannot be tolerated. The "wait" and "signal" operations on semaphores attached to tasks allow for event servicing by the time it occurs. The "send" and "receive" primitives allow for exchange of data when available. As long as no event has occured that would cause a blocked process ("wait" was performed on an appropriate semaphore) to be activated (involving the "signal" operation), the CPU is not paralysed by busy awaiting the next event to occur. Moreover, the mix of I/O tasks (data acquisition tasks, output), and CPU-bound tasks (control algorithms) ensures efficient processor utilization.

2.2.3 Programming languages for multi-task and real-time applications

The standard versions of the programming languages: APL, BASIC, PL/1, FORTRAN, COBOL, and C are not tailored to support multiprogramming and real-time applications. The language C, however, includes system calls for real-time applications as a standard feature. Special versions of C do support multiprogramming. As for the other languages mentioned, upgraded versions are available incorporating features for both real-time operations and multi-tasking.

One group of programming languages is referred to as interpreter languages (APL, BASIC). Another group is known as compiler languages (COBOL, FORTRAN, PASCAL, C). The real-time versions of the languages in widespread use are REAL-TIME BASIC and REAL-TIME FORTRAN. CONCURRENT PASCAL is due to its object oriented programming features a particular language mostly used for expert system design (Mellichamp, (1983)).

A third group of languages for special applications such as process control are known as table-driven languages. They are strongly dependent upon the central processor and hardware at hand. They are usually not portable across computers.

As for the real-time versions of BASIC and FORTRAN, as well as for the language C in general, all provide real-time macro commands. These macros, although syntactically different, provide system calls for:

(1) Input/output
(2) Task creation and deletion
(3) Intertask communication
(4) Overlay and special queuing
(5) Usage of the clock
(6) Task identification
(7) Task/operator communication
(8) Operator/task interaction

Especially real-time FORTRAN was developed according to the ISA standard S61.3 (Instrument Society of America), where the standards specify the procedure groups: "executive interface routines", "process input/output routines", "bit string functions", "random file handlers", and "task management routines".

Examining the features of several languages that proclaim supporting real-time and multi-task programming will lead to the conclusion that each system call suggested by the ISA standard S61.3 has a corresponding procedure provided by the language.

However, subtle differences among the languages may be important for certain applications. This is discussed next.

2.3 Suitability of Commercial Software for PC-controlled injection-moulding at McGill

The following constraints affecting the design of a PC control system for injection moulding are known. Injection moulding requires fast processing speed of the control system to be established. Especially the injection pressure rises fast. I/O of data and computation of the control output has to happen virtually instantaneously. A time-precise sequencing of the machine cycles is required. Various and numerous changes of state of the machine have to be serviced virtually immediately. Temperature control, coolant flow-rate control, and injection-pressure control may be performed simultaneously. Control schemes are not straightforward and need to be frequently modified. Logging of data applies to multiple input channels. All features have to be configured user-interactively.

This variety of tasks identified above, implies the establishment of a program hierarchy composed of several modules. Virtually any level of the computer system has to be accessible to a system developer for customisation.

For this purpose, a versatile multi-tasking operating system and a low-level program

ming language for system development are necessary. There are some considerations as outlined on the following pages which deserve attention before making a choice .

2.3.1 Real-time and multi-tasking operating systems

QNX (version 2.1 & 4.0) and AMX for the INTEL 86-family of chips, as well as PDOS for MOTOROLA 68000 16/32-bit-processor based systems were designed for real-time multiprogramming. All of them feature a small sized and thus rapid system kernel in charge for interrupt handling, dispatching (scheduling), task synchronisation (semaphores), task suspension, message passing and memory allocation. Additionally, buffered I/O is supported and the 64K barrier of code size for large applications can be broken, since a full 32bit memory addressing mechanism is supported on Intel 80386 and MC 68000 machines. Finally, all systems perform round-robin scheduling which can be stripped to first-in first-out scheduling on request.

PDOS provides libraries for the languages: FORTRAN, BASIC and C, whereas QNX and AMX are restricted to the use of C for system development.

It is not obvious that these systems differ in performance. However, subtle differences might be detected when actually using the tools for system development. Secondly, syntactical differences for shell and language commands may influence the inclination toward a product.

QNX was given preference. Firstly, this is because a UNIX-style shell is maintained,

which is considered to be a future system-shell standard on whatever machine (as defined by the POSIX standard of IEEE). Secondly, C compilers are available for QNX, that for most functions, adhere to the ANSI standard released for the language C. Both features of QNX will facilitate future migration of programs to other computers.

It was not taken into account that QNX provides features to support a network of computers (LAN). This feature counts when any potential expansion of the single PC (console) application to a workstation is intended to be developed. This may include one PC for graphical-display and user-interface purposes, and a second PC for control related tasks.

2.3.2 Real-time programming languages

If the PDOS operating system were selected, the programming language of choice could either be real-time BASIC, real-time FORTRAN or C. Execution of a program hierarchy written in BASIC would involve the BASIC interpreter, which stays resident in main memory, statement by statement. Once a statement's execution were under way, interrupts would be disabled until completion, and enabled right after. An event that occurred in the meanwhile would have to be held pending until serviced. Additionally, the scheduling mechanism reacting on interrupts would similarly be suspended. Thus, state changes of tasks could not be acknowledged, and no immediate scheduling decision could follow until the statement were completely executed. For applications that show fast changes of a process's state, this toggling between interpreter and scheduler cannot be tolerated, which means PC control would certainly fail ! Modularization of programs is strongly facilitated if global storage of data in a program is possible. Although BASIC does not support such a feature, the PDOS version has an

extension to the language for limited global storage.

Compiled programs do not invoke an interpreter that links the required system routines to every statement at execution time. At run time compiled code is readily written in machine code. It is immediately executable. This enables faster processing compared to interpreting on one hand, and has the advantage that interrupts are not disabled during execution of a statement. This can be considerably long when measured on a micro-processor time scale.

Differences between real-time FORTRAN and C are by no means obvious and might not even exist if performance is considered only. However, the syntax a programmer has to comply with may prove more convenient in the case of C. FORTRAN supports global storage of variables by means of the "COMMON" statement, whereas C simply requires the variables affected to be declared outside any function, including 'main()'. On the other hand, the system development features of C surmount the capacity of FORTRAN.

Table driven languages simplify programming, since only little knowledge about language syntax is required. However, due to this ease of usage, interpreting overhead is incurred, which slows down execution. This might be a bottleneck for time-critical applications apart from the fact that no standards for table driven programs exist. Portability thus suffers, and migration to future hardware and software is virtually impossible. Table driven languages thus tend to be very machine specific.

2.3.3 Conclusion

As an overall conclusion, the decision of acquiring a QNX operating system and a C compiler for development of a system incorporating a 32bit Intel 80386 central processor may well be justified considering the arguments above. This system is to achieve PC- control for injection moulding. Its potential performance and the requirements for large and time-critical applications, which fully apply to injection moulding, were proved to match well.

2.4 System components for PC-controlled injection moulding at McGill(1991) and limitations

2.4.1 IBM & ANALOG DEVICES hardware components and constraints on memory size

A personal computer, IBM PS/2 Model 70, was chosen to perform both the control system tasks and the user-interface tasks. It was originally equipped with 2 Mbytes of memory. Its memory (RAM) was extended to a total of 8 Mbytes, since the forecasted acquisition of large amounts of process data will require a suitably sized circular input buffer.

Estimating memory requirements, at a sampling rate of 1 ms one thousand readings will occur per second. This is essentially needed for pressure control during injection (less than 2s) that features a fast change of mould pressure. Accounting for 16 input channels (see below), this results in 16 Kbytes of memory occupation per second. This amount, finally, has to be doubled for each integer read, yielding 32 Kbytes per second of data to be read.

In the case cache-memory (read-ahead into input registers of RAM and write-after to disk) would be extended to 6 Mbytes, and assumed a complete machine cycle would last 20 seconds, data of roughly 9 subsequent cycles could be stored. This applies if only capacity constraints were considered when sampling at a rate of 1ms.

However, data acquisition at that speed will not be maintained for complete cycles. On one hand, dynamic memory management and task switching between interface and control algorithms could block both memory access and CPU time for considerable slices of time. On the other hand, slower process dynamics during the remaining phases after injection do not require fast sampling. The fine tuning of memory management, sampling rates, algorithms, and displays subjects to testing and might induce a modification of the system configuration. The computer's hard disk has a capacity of 60 Mbytes. Graphic capabilities for terminal output are provided by a VGA card (640x480 resolution, 16 colours).

The data acquisition sub-system, including analog-to-digital and digital-to-analog conversion, is composed of two Analog Devices RTI-220 boards and a single Analog Devices RTI-217 board for digital input/output. The initials abbreviate the notion of real-time interface. Each RTI-220 board provides 16 (channel 0 through channel 15) channels for analog input and has 4 analog output channels. Further technical details are contained in the manufacturer's reference manuals.

2.4.2 QUANTUM COMPUTER INNOVATIONS, WATCOM Software and constraints on program development

For establishing a preliminary version of the user interface and the data acquisition tasks the C-86 compiler and linker package for C-language applications written by Computer Innovations was used. It was developed for application programs run under QNX (see below) on PCs which incorporate a CPU of the INTEL 8086-family. Except for writing interrupt handlers, C-86 offers versatile routines for system configuration. Its low cost is one of the most striking features. Due to poor documentation, much coding required a trial-and-error approach. C-86 was used to develop a preliminary version of the user interface for QNX 2.15 (see below).

For real-time applications on PCs with limited resources, saving of memory and execution time matters a great deal. Reducing the size of machine code and optimizing it (e.g. expelling not alternating loop variables outside the loop) could be achieved at a later state of the project by harnessing the capabilities of WATCOM C, which is a compiler and linker package for applications on QNX 4.0. WATCOM C was the compiler used in the final stages of this project.

QNX is a trademark of Quantum Software Systems Ltd. for its real-time multi-tasking operating system. Both versions QNX 2.15 and QNX 4.0 were used to develop the user interface tasks to be found on the diskettes included. Development was started with QNX 2.15 but switched to QNX 4.0 which was recently released. Especially QNX 4.0 offers a versatile system shell that closely matches the standard set by UNIX. This will count considerably once portability of the system to future platforms is considered. Currently, at University of California at Berkeley, the development of a real-time UNIX operating system is under way.

is estimated to continue affecting PC levels. Some non UNIX related shell commands and system calls do meet the POSIX standard instead. A timing advantage of QNX 4.0 compared to QNX 2.15 could be demonstrated on simulations, and thus justified the decision to finally migrate from version 2.15 to version 4.0. The reason, QNX 4.0 runs faster derives from a differently organised system kernel (presumably, it only runs 16 routines written in assembler).

3. **Major interface design specifications and features on the system level**

The tasks that make up the user interface, their identity (system character strings that serve as names), and their size in kilo-bytes of machine code (specific to QNX 4.0) are given in the table below.

	TASK	SIZE (Kbytes)	SPECIFICATION
(1)	'imm'	219 (at runtime)	Interface Main Task
(2)	'statdip2'	49	Status Display Task
(3)	'viewdat'	64	Dummy Task for Sensor Reading Display
(4)	'variable'	13	Simulation Task for Data Acquisition and Control Tasks
(5)	'barreltemp'	25	Real-time Display of Barrel Heater Temperatures

All tasks (modules) were compiled in the medium memory mode, which reserves a single segment less than 64 Kbytes for data and several segments larger than 64 Kbytes for machine code. As a result, the interface related tasks do not exert a strong constraint on system resources and should leave much of the CPU time free to other tasks once the appropriate scheduling policy was adopted (see section 8.2).

To keep memory occupation small and processing speed fast only so called terminal functions that directly access the console's video adapter were incorporated in the interface code. The screen thus operates in text mode which is determined by exclusively referencing the extended ASCII character code table, and by dividing the screen into 80 columns and 25 lines. Screen input and output does not require major computation, since mere addresses of the ASCII data base and colour attributes (hexadecimal numbers) stored in the

video registers are shifted back and forth. One screen requires 80 · 25 · 2 = 4 Kbytes of (virtual) memory (2 bytes per screen character, 80 chars. per line, 25 lines).

Given this relatively low claim on free memory, many screens and warnings were devised for both casual and professional users. Casual users will find the system ready to use and are thus not likely to encounter messages that help a professional user find problems with the interface program-environment. Any potential user might most frequently encounter messages which notify that a wrong key was pressed. In contrast to a casual user, a professional user could receive messages dealing with operating violations such as:

(1) Timers are set to zero and the machine was requested to run;
(2) ADC channels were reset to zero, such that no channel is currently specified;
(3) Files to be loaded do not exist or the path specified is invalid;
(4) Requests to exit without having saved the data;
(5) Typing and range errors, which were detected by the error checking routine mentioned above;
(6) If the menu driven set-up were forgone or a corrupt set-up data file was loaded, and the machine were requested to run;
(7) Sending and receiving of messages and starting and halting of tasks failed;
(8) Allocating memory if too many segments are occupied, which could occur upon subsequent calls of the interface task;

The specifications outlined thereafter pertain to the priority distribution among all tasks to be run. Secondly, the interface features for different types of users are explained in the following paragraphs.

3.1 Handling interface priority versus priorities of control system and data-acquisition related tasks

It will first be mentioned how scheduling works by default. QNX 4.0 offers a choice of scheduling algorithms that can explicitly be specified at runtime. The default scheduling policy is referred to as "other mechanism than round-robin or first-in first-out scheduling" [descending priority scheduling (priority decay)]. Once a task is created its process descriptor is placed at the tip of the ready queue. It is the next task to be running after the time slice for the currently running task has expired. The priority assigned to it will be reduced one step towards its base priority (10 by default out of a range from 1 to 29) if it was not reached yet. Assuming it was not reached, it will be placed back into the processor queue after the first time slice of allowable CPU time for it has expired. Due to its lower priority it might then not be the first non-running job in the queue. It was placed into the queue according to the order of current priorities of all the waiting processes in the queue. The higher the priority of a process the closer its descriptor will be placed to the top. The more the task actually runs, the further its priority is lowered so that it finally finds its place in the rear of the queue, once its base priority was reached. There, it will eventually have to wait for long depending on the number of concurrent tasks to be run. This is a mechanism that allows for a propagation of tasks in the queue, and thus affects the sequence of CPU switching.

At the present stage of the user interface and control tasks project four main tasks are to be run concurrently. The interface main task will start a second task for real-time status display of the injection-moulding machine. Right before, a third task that starts all future data acquisition and control tasks must have been started. This is up to the present simulated by a dummy task. Finally, the interface task will create a fourth task that upgrades the lower screen,

monitoring the current barrel-heater temperatures.

The present version of the user interface does not include complex features for changing the scheduling policy to round-robin and omits the setting of individual priorities. However, when all the control tasks could finally be added to the system, WATCOM C provides runtime routines to adjust for an appropriate change (i.e. 'qnx_scheduler()', and 'set_priority()'). Code must be added to the interface main module ('if_start.c') and to any other stand-alone task that may require a priority boost. As for the user interface, it is considered to provide the appropriate code in the shape of commented lines (/***... expressions ...***/) that have to be reactivated when required.

Up to the present, the simulation of the control and data acquisition tasks did not lead to a bottleneck concerning CPU switching invoked by the scheduler. All computations at the simulation level turned out to be quite simple (i.e. 8 concurrent floating point operations). Refer to section 8.2 where a change of the scheduling policy is outlined.

3.2 Providing features for professional versus casual users

According to the production and research perspective (refer to section 2.1), changing the system configuration will either be requested or will be omitted by a user.

Professional users might appreciate the following features. Menus are provided, by means of which parameters can easily be changed. For safety reasons, both typing errors and range violations are immediately traced. Moreover, for transparency reasons, warnings are printed with directives to remove the error cause. The error checking routine detects space gaps,

invalid characters including a decimal point where an integer is required, and total gaps (spaces). It furthermore prints the maximum and minimum allowable value on screen whenever one was violated. Nothing has to be typed explicitly except for filenames. Options are provided and can always be scrolled by moving either the cursor upon items or by pressing the arrow keys.

In order to facilitate the work and usage for casual users, all that is left to do on a session is to load a set-up data file and to press the key <enter> to proceed to the monitor that awakens all the other tasks and the machine action. Two keys only have to be pressed to bring the main menu back. Wrong keys pressed inadvertently will make an alarm ring to attract and guide the operator's attention. In the case a casual user changed a default set-up parameter and tried to get back to the main menu to proceed, a second-level error- checking routine would intervene if one of the error conditions above held. It would block further progressing and force the operator to get back to the faulty item or to exit (safety barrier).

For professional users the objective was to allow them to govern the system at basically all levels. Error checking (except for typing) can thus be switched off. All parameters including the minima and maxima can be edited using a full-screen editor. All pages are defined in a way that future additions can be made, to a certain extent, without rearranging the whole editor. However, the editor is optional for set-up parameters, since all sub-menus can be used to define a new set-up data file. Minima and maxima can only be defined through means of the editor.

An important interface design-objective pertains to a graphical display of sensors, the locations of these sensors, and current readings. The real-time display of current sensor readings on screen is supposed to be a future feature. Accounting for future interface modifications, the logical branching for the display of sensor readings in the interface main-menu was considered.

4. Interface structure and features on the user level

4.1 Top-down menu levels

The menu structure common to commercial software was adopted. Starting with activating an item of the main menu, other menus would pop up on screen, monitoring another set of items which could similarly be activated. Generally, the highlighted leading characters of menu items could be typed in upper case or in lower case to invoke the appropriate sub-menu. Alternatively, a movable bar, inversely coloured than the items, could be positioned on the items using the cursor keys. Pressing the key <ENTER> would then cause activation. This design principle is used throughout the interface leading to a hierarchical menu structure (see appendix, Illustration of menu levels {index FC0}).

From within the main menu the following items can be activated:

(1) A menu of options needed to load and edit a data file;
(2) A menu of options to specify further menus needed for machine set-up;
(3) The editor for set-up data and range data files;
(4) A task simulating graphics for display of sensor readings;
(5) A monitor for control system configuration, machine activation and manual operation;
(6) A menu to specify filenames, slaying the upper right clock, and to set the error checking mode;

On the level of set-up parameters (see appendix, Flowchart 2 {index FC2}) menus are provided to specify:

(1) Timers for injection, holding, cooling and resetting the barrel;
(2) ADC parameters, including sampling rates, ADC channels, and the ADC environment;
(3) Controller parameters for standard PID controllers, 4 assumed;
(4) Start cycle and stop cycle for ADC and recording;
(5) Barrel heater set points, 4 are provided;
(6) Several dummy parameters to be activated at a later stage of the project, i.e. variable and fixed set points, a factor to change the upgrading speed of the status display, and miscellaneous timers;

The monitor for control-system configuration (see appendix, Flowchart 4 {index FC4}) provides options to specify:

(1) The controller type, if PID, Dahlin, Vogel-Edgar or GAO/Patterson Attenuator;
(2) The total number of cycles;
(3) The operating mode; In manual mode a menu pops up that activates 7 function keys for invoking one of the desired machine actions at a time (open mould, close it, move barrel forward, retract barrel, advance screw, retract screw, and halt all movements; after each command the system will accept a strike of the space bar to halt all movements (emergency abortion).
(4) If and when to start the real-time status display task in full automatic mode;

Other menus that involve the entry of filenames are the one providing options for loading a file (see appendix, Flowchart 1 {index FC1}) and the one for saving a file (see appendix, Flowchart 7 {index FC7}). Both menus are popped up on screen following calls from within the interface main menu or the editor main menu. Default filenames can be changed by calling a particular menu (see appendix, Flowchart 5 {index FC5}) from within the interface main-menu (i.e. item 6, 'initial settings').

4.2 Editor for default set-up data files & mini-max data files

The editor main-menu prompts the user to chose between editing a file containing set-up parameters or a file storing the appropriate minimum and maximum values (see appendix, Flowchart 3 {index FC3}).

Whichever option is chosen, the first menu to be active is the one to load a file that either exists or could be created if not. If it exists, the data could be loaded or all values could be reset. This happens in RAM only and nothing is written to the file unless saving was invoked. Menus including directives to guide the user through the file system will be popped up in descending size. The smallest window that is popped up at any given time is the only one activated. Upon completion of any command to be selected, these windows disappear in reverse order.

For both types of data, several screen pages are provided that subsequently present the values to be specified. Pages can be switched back and forth while showing the most recent values entered. When the last parameter on the last page is specified, the user is automatically

prompted to save all data entered, to exit or to jump back to the interface main-menu. Only for the optional editing of range data files is typing error protection is provided.

Corrupt data inadvertently included in set-up data files will be discovered at the safety barrier before calling the machine activation and operation monitor (i.e. main-menu item 5, 'run & stop monitor'). Appropriate messages indicate the menu location of the corrupt value, and the key to get there.

It is always possible to reactivate the default value of a parameter that was either stored on an existing and loaded file or that is provided by the editor if no file was loaded. Furthermore, it is possible to step back through the pages, item by item, which will automatically reset the values encountered to default. In this case no menus, windows, or messages will be popped up.

When editing and saving of a file was terminated, a small window presents a directive and the options to either edit the same file starting at the first item, to edit another file of the same category, or to switch category and edit a particular file.

It is always possible to leave the editor and jump back to the main menu from the first item on the first pages in both editing modes.

4.3 Error checking

It was previously mentioned that error checking is provided. It occurs in two modes and on two levels (see appendix, Flowchart 6 {index FC6}). By default, the checking routine is bound to detect both errors related to corrupt characters and violations of maximum and minimum allowable values of machine variables, for example barrel temperatures.

The flag set to invoke this checking mode can be deactivated from within the initial-settings menu which is accessible via the interface main-menu (i.e. item 6). Range checking can thus either be on or off. Whenever reaching this point of the interface, any user will be forced to use the down-arrow key to scroll at the item. Only this key will be accepted at this instance. After having specified if checking for range violations is to be on or off, appropriate function keys bring back the interface main-menu or exit.

It must be mentioned that all parameters are character strings on all the levels pertaining to loading, editing and saving. Conversion to the appropriate types, which are either type integer or type float, is accomplished at a later stage. Logically, conversion will always be performed before the parameters are being stored into global memory.

The steps that check for corrupt characters invoke functions that subsequently chop single characters off the string. Each character is then compared to a set of allowable characters. Spaces are checked conditionally, meaning that a warning is raised only if a space gap between two valid characters was encountered. Any error of the kind of invalid character causes a window to be popped up that includes a warning and a directive for further progressing.

In the case no invalid character of the string could be detected, it will be converted to either type float or type integer. The following steps detect if the converted value either exceeds the allowable maximum or falls below the allowable minimum. If one of the error conditions holds a message will be printed on screen in a small window.

The second level of error checking is located ahead of the first level in the interface code sequence which was described above. Every parameter will be checked alike for corrupt characters and magnitudes if the flag for range checking was set. Assumed no set-up menu was previously called, which means a set-up data file could have been loaded and changes by typing were omitted, error checking at this level provides the only safety barrier before the machine can actually be run. In the opposite event, if set-up parameters were modified after having been loaded or were alternatively specified from scratch, range-error checking would be twofold.

However, if the checking for range violation was deactivated, at least corrupt characters would be sought after by the error checking routine. Again, this would affect both levels in the case parameters were typed or modified from within set-up menus.

In general, this error checking mode can never be deactivated, since it provides the only protection against invalid conversions from type character to type float or integer due to corrupt characters.

Furthermore, the loading and saving routines invoked by the set-up menus or by the editor provide writing or reading checks of the strings stored on file. On writing or reading error of an item, a window would be popped up on screen, indicating either reading or writing error. A reading error prompts the user to jump back to the load-options menu to induce a

change of the filename. It is likely that inappropriate files are requested to be read. Error on saving an item equally causes a warning to be printed. In both cases, exiting the program is optional.

Finally, all C-language functions that affect the allocation of memory and the initialisation of either global or local pointers invoke a warning to be printed if they did not complete correctly. One case that is likely to occur during testing is the attempt to subsequently start the user-interface task. If the cache memory is too large and other memory segments were not freed, which can occur if other tasks were still resident in memory, it is likely that no free RAM remains to hold the code for the interface task. A message will notify the user that no memory is available and exit. With the help of system routines (i.e. 'sin' and 'slay') this problem can be cleared.

4.4 Background real-time machine-status display

After the machine activation and operation monitor was activated from the main menu, the real-time status display task will be created ('statdip2').

It first receives a message of global pointers which address memory locations of timing counters for each of the stages. According to the time elapsed during each of the stages, the contents will be counted upwards by the tasks in charge for data acquisition and control. The display task will read these memory locations in a timed manner which is achieved by attaching a QNX timer. The reading rate is currently set to 5 milliseconds and can presently only be changed by modifying its source code. At a later stage of the project it is proposed a factor

to be specified in a menu to customize this rate.

The display task receives a second message, containing arguments that specify the screen position, the time intervals of the four stages, and the colour attributes. By changing initialisation of these values, which happens within the interface main-module ('if_start.c'), the status display can be customized.

Once the sending and receiving of the two messages was accomplished, the status display task runs concurrently in the background with the interface task. As was mentioned before, scheduling is performed according to a policy that differs from round-robin and first-in first-out scheduling.

Currently, the time of each machine stage is displayed as a bar of a certain length which represents its portion of total cycle time. Each of the four differently coloured bars are logically divided into 5 smaller fractions. Whenever the CPU is ready to pay attention to the display task, it first reads the memory segment holding an identifier of the current stage. Afterwards, it reads the appropriate stage counter which holds an integer according to the time elapsed. The next step that follows is the comparison of that counter with a marginal integer number that indicates the total stage time. According to the ratio of the current counter and the marginal integer, the field will be reprinted stepwise in a colour which differs from the original one. For simplicity, say that each step corresponds to 20% of total stage time, which amounts to one field character. Assume the total field length were 5 characters. If then 20% of the stage time elapsed corresponding to a counter/margin ratio of 0.2, the first field character would be reprinted in a colour say from highlighted yellow to blinking red. The remaining field characters would be reprinted accordingly as more time elapsed.

The field lengths vary, since the time intervals for different set-ups change. For the cases that the field lengths differ from a total of 5 characters, a routine finds out the number of characters available to represent or leave out the five succeeding steps of 20% of total stage time. All these computations affected are undertaken once on task creation. It means that when the machine actually runs, the display task does not claim CPU time for any computations except for the few statements affecting the field reprinting mechanism.

Once a new stage of the machine is entered, the whole corresponding field is first highlighted. Time proportionate reprinting of the five field sections occurs next as described above.

The disadvantage of this method, which has its cause in the somewhat arbitrary division of the fields into 5 fractions, manifests as a red blinking colour bar that moves from left to right across the fields in a discontinuous manner.

The advantage, however, is short computation at any given time. Only a few characters are to be printed on screen. Repeating the most important constraint imposed on the interface development, it was necessary to leave as much CPU time free as possible. On the other hand, this technique required a more complicated and thus bigger code.

All the other information, which is the display of the stage time intervals, the current stage, the current operating mode, and the total cycle time is printed once. This is not subject to updating, and thus is not time-critical.

4.5 Control-System configuration-monitor and operating options

A menu which was previously referred to as run-and-stop monitor (see appendix, Flowchart 4 {index FC4}) for the machine will be printed on screen once all parameter conversions from type string to type float or integer were successfully accomplished. The safety barrier must have been successfully passed.

A program module, called 'fw_do.c', provides the source code. In the beginning, kernel functions will be executed that subsequently serve to create all tasks that make up the system.

Control system configuration and the setting of the machine's operating mode are required to be specified by scrolling options displayed in particular fields on the screen. Currently, only the controller type can be specified. Typing is involved to specify the total number of cycles the machine is requested to run. If the flag for checking range violations was set (default), both typing errors and magnitudes of the converted value will be traced by the error-checking routine (refer to section 4.3).

At this point of execution, a function will be called (i.e. module 'f_fromui.c') that allocates and initializes global memory segments for parameters to be shared with the control system and data acquisition related tasks. The idea is that the exchange of data subject to changes made by the user during a session with the system is done without the obligation to send and receive messages. Otherwise synchronisation between the tasks, which is challenging, would be required, since the exact time for data transfer is undetermined. Global memory has the advantage that tasks can refer to the data stored following the rule "grab it when you need it". Send and receive blocking is not involved, and so facilitates time sharing among the tasks.

However, one single send and reception of these global pointers to the appropriate tasks is required at the starting point of execution. Each task that has to share global data with other tasks needs to know the memory locations. This cannot be avoided.

After the initialization of the global pointers, the current values of the parameters to be shared are assigned to them by means of the redirection operation in C (asterisk: *pointer=value).

The following code sections are concerned with sending and receiving of messages. This module, 'fw_do.c', will receive a structure (i.e. struct glob3, defined in 'shared.h') that contains global pointers to the stage counters (see section 4.4 above). This structure was sent by the control system and data acquisition related tasks, which allocated the global segments. The initialisation of the pointers holding the addresses of which preceded. For identification as the target task to which this structure was sent, the interface task has to attach a name, which is "ui" to abbreviate the notion of user interface.

Subsequently, the structure glob3, which was just received, will be sent to both the status display task ('statdip2') and to the task in charge for monitoring the current barrel-heater temperatures ('barreltemp'). In addition, the structure 'glob4' will be sent to the latter task by the module 'fw_do.c'.

By means of checking the flags 'send_???' (??? represent symbols indicating which structure was sent), sending and receiving of messages is accomplished once for any given session. So is the creation of all tasks that make up the system (see below). As for the present state of the interface, only those parameters can be modified that are accessible via this

monitor. These are the identifiers for the controllers, the total cycle time, the operating modes, the parameters that represent the manual operating commands, and the parameter to indicate starting and stopping of the machine.

Leaving the monitor will invoke the operating system shell to slay all tasks that are not related to the interface task. The flags 'send_???' will then be reset. Any subsequent entry into the monitor will cause the same statements to be executed as described above. Sending and receiving of the global structures, as well as succeeding task creation is invoked again.

In the case manual operating mode was chosen, the upper half of the screen will be reserved for a special menu that displays key definitions and directives needed for manual operation. Each command executed will cause a coloured bar to jump back onto the command field to halt the machine. Subsequently pressing the key <ENTER>, or alternatively the space bar or the character <H> in lower or upper case, would cause the identifier for machine halting to be written into global memory (i.e. by the redirection assignment: *fromui.ip_startstop=stop). It is meant that this location is often checked by the control system and data acquisition tasks to invoke machine halting.

Pressing another key brings back the sub-menus for controller specification, total number of cycles, and the operating modes (i.e. full-automatic mode, manual mode, or semi-automatic mode).

Operation in full-automatic mode prompts the user to start the machine by pressing a key. Subsequently hitting the space bar causes the identifier 'stop' to be written to global memory. This corresponds to the mechanism described above for halting the machine in

manual operation. However, no special key is required to bring back the sub-menus for controller types, cycles, and modes. This follows automatically after the space bar was hit. For redundancy and thus safety reasons, a special function key, <F10>, invokes exactly the same action.

This mechanism, i.e. the reprinting and reactivating of all sub-menus when appropriate keys were pressed, allows for switching the operating mode and control strategies for runs governed by the same set-up.

In general, once the operating mode was specified, all the other tasks would be created and run in the background from then on.

5. Source code architecture

5.1 User-defined header files: "colours.h, fl_colours.h, f_decl.h and shared.h"

Both the interface and the display programs are broken down into several modules. A listing of the modules is obtained by checking the contents of the diskette attached. The files featuring the extension ".h" are user-defined header files that are included in some of the modules.

The file "colours.h" contains a variety of display attributes for screen characters which are referenced whenever a terminal function prints on screen.

Another file called "colours_fl.h" is quite similar to the one mentioned above. It equally contains a set of display attributes. The difference, however, arises from the setting of the flag to flush the terminal functions after output. It was found that due to multi-task printing on screen this is a requirement to achieve concurrent printing. Only the child tasks that will be created at a later state need to refer to these attributes. Many of the printing routines will have been called already by the parent task.

For this project, the display tasks, i.e. 'statdip2', 'barreltemp', and 'viewdat', include the header file "colours_fl.h".

The parent task, which is called 'imm', also refers to the header file "colours_fl.h" at the point where the display attributes for the status display task are initialized (see module 'if_start.c').

In the view of the C-language, all modules are basically functions that are called either within the main module called 'if_start.c' or within other modules (nested programming). The module names and the corresponding function names coincide. Instead of explicitly declaring the functions (modules) referenced at the top of the calling modules, the header file "f_decl.h" is included in all of them. At compile time, the C-compiler will compare any function declaration with the syntax of the function definition placed in the appropriate module, i.e. the variable list that includes the storage class for all input and output parameters. Errors which emerge from function I/O violations can thus be detected and avoided at compile time.

The header file called "shared.h" serves to be the declaration of structures referenced by more than just one module. Instead of repeating the declarations, the compiler's needs can be satisfied by including this header file. This contributes to reducing source-code size and editing time. A structure is generally considered a new type. Variables can be declared of such a type, 'struct name', as commonly as it works with other types, namely: integer, float, and character. One line only is required for declaration, even if the structure declaration itself exceeds more than a screen page.

On the other hand, it helps avoid confusion of structure members which would be not easy to detect. The file "shared.h" includes the declaration of four structures named 'glob1', 'glob2', 'glob3', and 'glob4'. All of them will be sent as messages to display tasks from within the interface module called 'fw_do.c' (see section 4.5).

The structure 'glob3' is received from the task 'variable'. Right after, structure 'glob1' is sent to the task 'variable' by the module 'fw_do.c' of the task 'imm'. 'Glob3' is then sent to the task 'statdip2' and the task 'barreltemp' by the task 'imm'. The latter task also receives the

structure 'glob4'. The structures 'glob1', 'glob3', and 'glob4' include global pointers to shared parameters. Structure 'glob2' does not include parameters to be shared globally, and is also sent to the status display task 'statdip2' by the task 'imm'. It contains arguments for customized printing of the display.

5.2 Library object modules of 'if40.lib'

All modules that make up the interface tasks and the status display task are merged together into a library file. Throughout the interface program, function names coincide with the module name that holds the function definition, i.e. the routine 'function_name()' is defined in a file called 'function_name.c'.

Except for the module 'f_timesdisp.c' the C-routines were compiled in the medium memory mode (compiler switch: -mm, see WATCOM C-compiler user's guide). This indicates that code of the resulting executable file exceeds the size of 64 Kbytes, whereas a data segment of 64 Kbytes is sufficient to hold all data the interface handles. The routine 'f_timesdisp()' defined in the module 'f_timesdisp.c' is called within 'statdip2.c'. Both modules are small in size. The files 'f_timesdisp.c' and 'statdip2.c' were therefore compiled in the small memory mode (switch: -ms).

To obtain a listing of the object files that are merged into the library 'if40.lib' the WATCOM library manager provides a tool to save the listing on file. The command would be:

wlib -l='name' if40.lib.

Using the shell command:

more 'name' |less

would result in a pagewise print-out which is useful for screening. The library 'if40.lib' has to be linked to the interface main module called 'if_start.c' and to the status-display main-module called 'statdip2.c'. The commands to be typed could follow the syntax given thereafter:

cc -g -o imm -mm if_start.c -lif40.lib

and

cc -g -o statdip2 -ms statdip2.c -lif40.lib.

To replace a modified and compiled module one would have to type:

wlib if40.lib +- (or -+) name.o.

A new module could be added or erased by typing:

wlib if40.lib +name.o for adding

and

wlib if40.lib -name to delete the module.

The library itself was created using the command provided by the compiler interface:

cc -A if40.lib *.o

This would cause the creation of the library 'if40.lib' by including all object files in the current directory. If a module were changed, it would have to replace its previous version stored in the library. For relinking the updated library to the main modules one would have the option to type:

cc -g -o imm -mm if_start.o -lif40.lib

or

cc -g -o statdip2 -ms statdisp2.o -lif40.lib.

5.3 Global, local, and system variables

The first code section of the interface main-module called 'if_start.c' declares a set of variables. The whole set of variables is declared outside the main module which assigns global storage class to it.

The groups of global variables can be listed as follows:

(1) Set-up parameters of type integer or float
(2) Pointers to strings for set-up parameters (not converted)
(3) Pointers to buffers for default parameter strings

(4) Minimum and maximum values of type integer and float
(5) Pointers to strings for minimum and maximum values (not converted)
(6) Pointers to strings for default minimum and maximum values
(7) Global pointers to parameters to be shared among tasks
(8) Arrays of pointers to strings for menu items
(9) Pointers to screen (page) buffers
(10) Variables used as flags (logical execution control)
(11) Identifiers for machine mode, stage, controller type, ADC channels, and ADC environment
(12) Unsigned integers for buffer sizes

By checking the list of external declarations included at the beginning of the modules, it is straightforward to trace back the locations where the variables are declared.

Some of the modules declare a few local variables which are only referenced from within the same module. This mostly affects auxiliary variables, as for example integers for loop counts.

Task identification is achieved by harnessing the QNX functions that provide names for tasks. A task that attaches a name can be identified within the system by other tasks that search for that name. Names are strings and are known as system variables. They do not serve any other function than task identification. A task that is able to locate another task by its name will receive a process identification-number by the routine 'qnx_name_locate()' in return.

This process-id is required to allow the locating task to send a message to the identified process. On the other hand, a receiving task can obtain the identification of a sending task by locating its name.

The following names are currently being attached by the tasks:

(1) "variable" by the task 'variable' = control and ADC tasks
(2) "display" by the task 'statdip2' = status display
(3) "ui" by the task 'imm' = interface task
(4) "barreltemp" by the task 'barreltemp' = temperature display

5.4 Logical branching, returned values, and jump marks

The control flow through the program is directed by means of logical operations which identify returned values of either C keyboard-functions or interface routines. By means of the 'goto' and 'switch' statements in C, program continuation can be directed to any statement within the same module. The success of a stack-pointer long-jump invoked by a 'goto'-statement requires the proper definition of a label or jump mark.

Labels and 'goto' statements are widely used in the interface program. Especially after a menu item was selected or a function key was pressed, a jump to the call statement of a specific function is required. At points where user input is required, the returned values are checked by a set of if-statements. If none of the conditions hold, a jump back to the input routine is invoked. This prevents the program from dealing corruptly with an undefined condition at input. Forcing the program to jump back to the previously called routine upon an unidentified return is maintained throughout the interface program.

Once the routine 'term_load()' was called, which initializes the structure 'term_state' and switches off echoing on input, pressing of a function key causes the return of a 16 bit integer

number. The definitions of the keys are located in the 'qnxterm.h' header file.

Identification of activated menu items is accomplished differently. Menu items are defined as elements of pointer arrays of type character. Each element is a pointer to such a menu item, which is a string. The function that performs selection returns a pointer to the menu item selected. Subsequently, a string function compares the string to which the returned pointer points with each of the menu items maintained by that particular menu. In the case the comparison is successful, meaning the string to which the returned pointer points, and the menu item selected match, a 'goto'-statement will be executed. This invokes a jump to the program location which is preceded by the 'goto'-label.

The user interface main-module, which is 'if_start.c', contains a great deal of labels and 'goto'- statements. They occur in the order of menus and routines to be called to run the machine during a given session. The sequence is as follows:

(1) label "mall": location where 'f_malloc()' is called to allocate memory for the numerous strings;
(2) label "mscr": location where 'w_main()' is called, which prints the main menu on screen;
(3) label "cum" : location where 'curs_main()' is activated for main menu item selection;
(4) label "ini" : location where 'f_iniset()' is called to specify default file names and error checking mode;
(5) label "load": location where 'f_load()' is activated to load a file;
(6) label "mdf" : location where 'f_makedefdat()' is called, which invokes the editor for set-up data and range data files;
(7) label "vdf" : location where task 'viewdat' is created
(8) label "sscr": location where 'w_setup_main()', which is the set-up main menu, is called;

(9) label "cus" : location where 'curs_setup()' is called to activate item selection;
(10) label "tim" : location where 'w_timers()' is called to specify all stage time intervals;
(11) label "adc" : location where 'w_ADCmain()' is called to pop up ADC main menu;
(12) label"rates": location where 'w_ADC()' is called to specify slow and fast sampling rates for each of the stages;
(13) label "chan": location where selection of fast or slow ADC channels is accomplished;
(14) label"schan": location where 'fw_schannels()' is called to specify slow ADC channels;
(15) label"fchan": location where 'fw_fchannels()' is called to specify fast ADC channels;
(16) label "op" : location where 'w_ADCops()' and 'curs_ADCops()' is called to set ADC environment;
(17) label "vfsp": location where 'w_varfix()' is called to specify fixed and variable set points;
(18) label "spb" : location where 'w_bheaters()' is called to specify barrel heater set points;
(19) label "cp" : location where 'w_control()' is called to specify PID controller parameters;
(20) label "rs" : location where 'w_runstop()' is called to specify miscellaneous timers and cycle numbers for timing purposes;
(21) label "save": location where 'f_save()' is called to invoke saving of data;
(22) label "conv": location where 'f_convertall()' is called which causes all parameters strings to be converted from ASCII to integer or float;
(23) label"cconv": location where 'f_convert()' is called which converts stage time intervals to unity in the case a value below 1 is specified (display);
(24) label "rscr": location where 'fw_do()' is called which is the monitor for control system configuration and operating options;
(25) label "off" : location where the clock is created if previously killed and the screen buffer is released;

In general, the control flow of most operations performed within the interface program and its modules is channelled through the main module and branched from there (if-then statements).

6. Coding strategy

Not all design objectives were known in detail at the time coding was to be performed. Details were added, modified or erased in the course of the project's development. However, the major design objectives were defined.

Initially, the hierarchy of top-down menu levels was established on paper. All the menu items to be specified that were known in the beginning were listed and grouped together. Some considerations about colours led to the conclusion that menus belonging to the same level within the program hierarchy would have to have a similar lay-out. As a result, the total number of menus, the number of different styles, and the number of menu items per menu could be stated.

The decision to handle parameters on all menu levels as ASCII strings derives from further considerations pertaining to interface I/O. Especially explicit typing of values and pathnames was intended to be performed by invoking the editing C-routine 'term_field()'. Activation of function keys was to be accomplished by referencing the 'term_key()' C-routine. Both decisions helped identify the type and number of variables needed for set-up parameter specification and minimum/maximum data.

A strategy to best perform saving and restoring of screens and windows was equally sought at an preliminary stage of the project. The strategy chosen is described below.

A decision was made to stick to formatted file I/O of the strings. This was considered an advantage, since shell commands (i.e. 'more' and 'less') could be activated for checking of

the file contents.

Some efforts were spent on finding out the number and scope of each task to be run for user-machine communication. The control system and data acquisition related tasks were said to be simulated in order to provide independency for interface program development.

Concluding preliminary studies, issues affecting the program hierarchy, data inheritance of modules, the most suitable storage class of variables, inclusion of user-defined header files, and control flow mechanisms to be implemented within the program were tackled.

In general, actual coding was involved in all subsequent steps of program development. Attention was first given to the creation of the interface task. The display tasks and the simulation for the control system and data acquisition related tasks were coded after a workable version of the interface program was available.

The basic logic of some menus was programmed and compiled first. In a succeeding step, the 'if40.lib' library was created to include these modules which existed as temporary primitive versions.

For initial testing, the interface main-module was defined next and was linked to the library if40.lib. In the course of program development, variables affecting the creation and coding of new modules were subsequently declared and added to the main module.

For continuing testing, this required upgrading the library, recompiling the main module and relinking it to the library resulting in a new executable file. Further features were

incorporated into the interface program by stepwise enlarging the code size and complexity of the main module and the remaining interface modules.

Editor commands were widely used to extend the modules with functional code sections that feature similar performance but at different locations on the screen (in the menu).

Those menu modules, the names of which include a 'w' for 'window', were first edited. For simplification of a menu's function, a single item was initially included per menu to facilitate testing.

In the following step, the basic logic for menu item selection was defined. This involved supporting the movement of a bar which is inversely coloured than the menu items. Customizing the selection mechanism for the main menu and the set-up menu was undertaken thereafter.

Code affecting the specification of parameters was programmed next. All menus were completed by adding all appropriate items. This step involved the initialization of pointers to strings that hold the parameter specifications (strings) and the default strings for it.

It was found that both groups of pointers have to be of the same storage class, i.e. of global storage class.

In the subsequent step, efforts were spent on finding a mechanism to load and save data. This included the creation of a file-system environment that would allow for pathname specification as well as provide editing utilities and a set of warnings and directives for error

handling. This is to avoid file opening failures as a consequence of wrong pathnames or non-existent files, and reading or writing failures. A probable instant of the latter case would occur if names for range data and set-up data were confused. A different format pertains to both file categories. A technique had to be determined that would master the specification and display of default filenames considering that the path and the filename of the latest file loaded should be displayed on subsequent calls of the loading and saving routine. This was finally accomplished using two global string buffers of 26 characters each that hold a default filename and the name for the specified file. After termination of the loading routine the filename specified is copied into the buffer for the default filename. The saving routine can reference the same buffers when activated, since both buffers (pointers) are global, and will thus inherit the name specifications made at file loading time.

The need for a provision of default filenames assigned to set-up data files, range data files, and ADC data files emerged upon completion of the loading and saving routines. This induced the creation of a module that provides specification options for default filenames. Even these strings were decided to be of global storage class, such that their definition in the main module's leading section could be modified if desired. This requires re-compilation and relinking of the 'if_start.c' module to the library for another executable interface file.

Further attention to coding was by then given to the editor, since all parameters for which range checking had to be provided were known. The idea emerged that pagewise editing utilities for all parameters could speed up both set-up data and range-data file creation. This resulted in an editor supporting two editing modes, one for set-up data specifications (strings) and a second one for minimum/maximum data specifications (strings). Saving and restoring of pages was an issue that required careful pointer definitions. It was found that

global pointers (i.e. char _far *pointer_to_page_i) to buffers for the saved screen pages best support page switching.

All issues pertaining to error checking were then addressed following the programming of the editor. Some considerations about a suitable mechanism emerged while using the interface programmed at its then present state. Error checking had to be accomplished on two levels and in two modes (see section 5.3). A safety barrier prior to activation of the run-and-stop monitor had to be established in the event the use of set-up menus for parameter specification were omitted. Secondly, errors would have to be detected immediately after a menu item was specified. This led to the creation of the error checking routine and the determination of the syntax for its call. All input parameters are of global storage class. At a certain point in the program where checking has to be performed, all of them need be initialized by locally appropriate values.

The application of error checking could then be extended to cover editing in the range-data mode. However, it was considered to be sufficient that only the validity of characters would be checked.

It would not have made sense to provide magnitude checking for minimum and maximum data. Above all, this interface level was designed following the assumption that research users only would make use of the editing utility.

At this point of the interface development, the transition to the programming of the display tasks, as well as the control system and data-acquisition simulation-task was undertaken. A great deal of efforts was spent on parameter sharing among the tasks embedded

in the program hierarchy. The technique found was the declaration of pointers to globally accessible memory segments. On the level of the interface program a routine had to be created that would perform pointer definition and initialization. It seemed favourable to include a user defined header file that would store the declaration of a structure comprising these global pointers. It was anticipated that the simulation task would need to reference the same structure at the time it was to be coded.

As a step to temporary interface-task completion, a routine was coded that performs the function of the safety barrier. No matter if a check of parameter specifications were already accomplished, all parameters that need be shared among tasks (see below) would be checked for character validity first and magnitude second. The syntax developed for local error checking was adopted to assemble the overall code of this routine.

At this point a workable version of the status-display task was being coded concurrently and independently from the interface task.

It was evident that synchronisation of the simulation task and the already existing interface and status display tasks had to be achieved. Particularly version 4.0 of QNX requires the sending and receiving of messages for task communication. Task identification was found to be best accomplished by harnessing the naming and name location utilities of QNX. On the interface level this required a module that starts all the other tasks of the overall system, and that arranges and coordinates the sending and receiving of messages among the tasks (refer to 'fw_do.c'). Sending was supposed to supply the tasks with all global pointers to shared data (see section 2.5 for details). One message-sending method considered was to send display attributes over to the status-display task. Roughly versions of this module and the simulation task were

first established. The inclusion of the status-display task was excluded for the time being. In subsequent refining steps the command sequence was established that would accomplish: task creation, synchronisation, and communication of all tasks.

Finally, customizing the barrel-heater temperature-display task and the switching of operating modes within the user interface received further attention. It was made sure that all crucial background tasks would be created before the operating mode can be switched from full-automatic to manual mode. This method had to be provided to allow switching of operating modes and control strategies alternately without previously terminating and subsequently restarting tasks that run in the background. This ensures that the machine can be operated in both full-automatic and manual mode while sustaining the machine settings and configuration.

7. Debugging and results

The activation of debugging tools for application programs required the interpretation of error messages as well as warnings issued by the compilers and linkers used for program development The user manuals for the above software include listings that give a short explanation for potential error causes that can be associated with the messages printed on screen. The variety of errors encountered can be broken down into the groups of compiling errors, linking errors, and runtime errors that are common to the subsequent steps of program development, which are: compiling, linking, and running. Compiling errors were detected first before the recognition of linking errors and the latter runtime errors. Corrections to the code were made immediately after the completion of one of these steps failed to issue compiling and linking commands recurrently.

The multi-tasking features of the operating system QNX allow for user-computer interaction on several virtual consoles (screens and keyboards). Concurrent applications can thus be examined simultaneously by switching these virtual consoles. I/O activities of several applications would be observable virtually at a time on virtual screens which are private to each application.

The debugging tools shipped with the versions 2.15 and 4.0 of QNX ('SID' and WVIDEO, respectively) harness this feature. During a debugging session of an application, one virtual screen displays the source code of the examined application, highlighting the statement to be executed next. The results of the execution of a statement are printed on a second virtual screen when I/O is involved. It is possible to toggle between these screens frequently.

Since transmission and reception of messages influence execution of a single program run in the context of a task hierarchy, it was necessary to perform concurrent debugging of all the tasks that communicate with each other. The advantage of this technique is that program execution unfolds stepwise, since the debugging routine has complete control over memory and program counters so that it halts execution at the latest executed statement of a program. It can thus exactly be determined at what location of the code and which task caused corruption. This concurrent debugging of programs that run simultaneously in the background required a modification of the system configuration files (see user manuals for console driver adjustments).

To start a debugging session including the interdependent tasks: 'imm', 'statdip2', and 'variable', a total of seven virtual screens had to be managed by the operating system. Two virtual screens were reserved for source code and I/O results of each of these tasks as described above. The seventh virtual screen was set active to call system routines needed for operating system and task-status checks. For example, upon completion of a statement in one of the three tasks, the 'send()' procedure in another task could have failed causing the task involved to be "blocked". This would be printed explicitly on the seventh virtual console after execution of the system routine 'sin'. It was possible to toggle frequently between all seven virtual screens.

This method proved very efficient to trace errors based on corrupted memory addresses invalid messages and logical deadlock of the sequence of task creation and communication. However, a great deal of efforts was spent to run the debugging routine on single erroneous programs (tasks). This required the activation of three virtual consoles: one for execution of operating system routines and their I/O performance, one for the display of the program's source code by the debugging routine, and one for the I/O activities of the examined program.

The most critical and tricky errors encountered after it was attempted to compile the source code of an application, after a trial to link related object modules to make an executable file, or after attempts to start the application from the shell are described next.

7.1 Interpreting compiler and linker error-messages

The compilation of an erroneous source file led to printing of a list of error messages. This required an immediate interpretation of a possible cause and editing of the source file to correct it. In practice, the switching between virtual screens proved to be greatly convenient for editing and iterative steps to compile a source file because of the following: on the first virtual screen, all the error messages could be listed and held, while editing could be performed on a second virtual screen with the help of a full-screen editor. Frequent attempts to re-compile such an erroneous application required frequent toggling between virtual consoles. Due to the buffering of keyboard input, the commands to initiate re-compiling could be repeated simply by pressing the arrow-keys, while the full keyboard definition for the editor was sustained on the other virtual console. It was therefore not necessary to repeat time-consuming typing of the same command seqences.

It was considered that one single coding error can cause several error messages to be printed that are not ultimately related to this coding error. This occurs due to propagation of this error through the compiling procedures. In fact, one coding error affects all interdependent routines of the compiler which thus fail and cause additional error messages to be printed in the order these routines are executed. Thus, the first message printed was acknowledged. Based on the error cause it suggested, the source code of the program was corrected using the editor.

The most tricky errors on the compiler level are references in the source code to invalid memory addresses or computations involving such invalid addresses. These cannot be detected by the compiler, since only a check of code syntax and variable declarations are performed. Actual values of variables including memory addresses are not verified by the compiler. Moreover, no messages are printed that can serve as a hint that program execution is at stake should linking be successfull and the executable file is attempted to run. Operations on invalid addresses certainly result in a fatal error at runtime which forces the operating system to terminate the application.

Likewise, some errors related to both compiling and linking do not cause messages to be printed immediately. This particularly pertains to the option of choosing a memory model for the object file to be generated by the compiler. WATCOM C offers a choice of memory models ranging from small to huge portions of memory to be allocated to an application program. This influences the amount of memory address space to be occupied by both data and code (see manual). The purpose sought by WATCOM was to offer program developers an opportunity to optimize the claim on free memory of applications. The choice of the memory model for a program at compile-time can interfere at linking-time with the models chosen for object libraries or other object files to be linked. The models of all constituents of an executable program must be consistent. Even if compiling of a source file was successful and no error messages were printed by the compiler, the linker can respond printing error messages when memory models of the object files to be linked mismatch. In this case, references to invalid memory locations are made by some sections in the object files.

It was attempted to link the 'if40.lib' object library to the main module object file 'if_start.o'. Inconsistencies between the claim on free memory for the executable file 'imm' to

be generated and the small memory models chosen for the object modules in 'if40.lib' and for 'if_start.o' led to linker error-messages. The strategy pursued after first occurrence of similar problems involved the compilation of all modules in large memory mode. This mode is defined to claim both for executable code and data memory segments larger than 64 Kbytes. After linking was successfully accomplished, it was possible to check for the size of the executable files created with the help of system routines. More information about the task's claim on memory could be obtained by actually running the task at hand on one virtual console, and then screening memory occupation on the second screen. The information printed could then be used to accommodate the memory layout chosen for the executable file 'imm'. All modules were re-compiled using a smaller memory model. In the case of the interface task 'imm', it was possible to switch the memory model from large to medium, which still refers to code larger than 64 Kbytes but claims only a single segment smaller than 64 Kbytes for data.

7.2 Interpreting run-time errors

Runtime errors of two categories were mainly encountered. One type of error did result in immediate program termination and the printing of a system error-message. The other type did not immediately terminate the application. The latter initially resulted in an undefined printing on screen. Upon pressing of keys, the application would terminate, causing the same system error-message to be printed as in the former case.

The reason for runtime errors of the first type are references to memory addresses that are invalid. Runtime errors of the second type can be explained by assuming that a shift of memory addresses has occurred after the completion of an illegal operation. The most common

coding errors encountered with regard to a similar shift were expressions that assigned a value of a global variable to a local variable. This phenomenon particularly held for global character strings and local strings. Therefore, corrupted and undefined printing was detected. To an extend that cannot fully be explained, such a shifting occured, once a child task was created from within a parent task. It is plausible that addresses of global variables are protected, such that they cannot be shifted, whereas the addresses of local variables are not protected. They can thus shift when some system activity affects partitioning of memory such as the loading of tasks into RAM. The shifting of addresses could be observed employing the above debugging technique with multiple screens.

Some runtime errors had in common that task creation and message transfer was not completed successfully. To send a message to a task that terminated before or awaiting a message from a task that terminated in the meanwhile causes the return of -1 to the system routines 'Send()' or 'Receive()'. This flag could be used in the interface programs to initiate printing of a message on screen if -1 was detected. However, these messages are user-defined. The system itself does not issue such messages.

7.3 Task simulation for detecting non-corrupting logical errors

It was previously mentioned that once a workable intermediate version of the interface task was available, tests were conducted. Tasks were created for testing that were used as expedients to represent later runtime versions of these tasks. Most of the modules that were subsequently added to the interface were tested independently from other modules. Instead of keeping the corps of the interface main module 'if_start.c', a module called 'test.c' was created

that had only the global parameters of 'if_start.c' copied into it. These are referenced by the modules. The call to a new module was in most cases the only line held in the body of 'test.c'. This provided a direct means to detect logical errors in conjunction with the debugging routine and the multi-screen technique.

To simulate the runtime version of the task 'variable' a provisory version was created registering the same name 'variable'. The simulating version accomplishes short hand what would be the case at runtime provided by the runtime version. To summarise, naming and locating names of other tasks, inter-task communication between these tasks, as well as memory allocation for shareable global storage was simulated. Upgrading the locations referred to as cycle-time counters for each of the four machine phases was achieved by running an infinite loop which was subsequently halted by calls to a C-language function ('sleep()') that suspends execution.

8. Conclusions and Recommendations

The conclusions given below focus on the methodical approach that led to the current workable version of the user interface for the PC-control system for injection moulding at McGill (1991). Upon closing this thesis, recommendations for further work are given that are to anticipate improvements concerning the development of the user interface towards a professional version.

8.1 Conclusions

The development of the user interface was an attempt to establish a custom system of software constituents tailored to predefined specifications. The pool of commercial software for program development was readily available from the start. Likewise, the computing facilities on which the interface program had to be created and was to be installed were ready for use. Furthermore, the user interface for PC controlled injection moulding was considered necessary as a result of plans to proceed in an on-going project concerned with the development of the control system. The method estimated best suited to master the interface development facing the above conditions was to subdivide the work to be done into a suite of phases namely:

phase I: acquisition of knowledge
phase II: definition of interface specifications
phase III: programming and iterative revisions
phase IV: final testing and inclusion into the overall system

It was evident that the lion's part of work had to emphasize on the acquisition of preliminary knowledge on a multitude of levels. The typical measures undertaken during the phases and the identified problem areas on which special efforts were spent are given in greater detail below:

(i) PHASE I - Acquisition of knowledge

Injection moulding

The injection moulding machine was inspected first and its operation was studied carefully. Information about process specifics were obtained by studying manuals and technical descriptions for injection moulding.

Problem areas

It was not possible to operate the machine without guidance to gain first hand knowledge. Modern injection moulding machines could not be examined to estimate a potential variety of functions that could also be more complex than those seen.

Operating system

The manuals were inspected carefully and demonstration programs were executed. Equally, it was tried to perform the installing procedure from the beginning. Shell commands were executed and the results were memorized. The configuration features were manipulated to identify subtle differences and to gain more insight of the management of system constituents such as memory management scheduling and the file system.

Problem areas

It required extensive reading to understand the interaction of the components. The reference manuals can only be understood if the organisation of a modern PC is

understood in detail. This required referencing appropriate literature which was difficult to find due to the high degree of recentness concerning the development of the the INTEL 386 CPU and multi-tasking operating systems. Limitations of the I/O procedures and memory protection are complex and crucial issues that need be understood to establish a program hierarchy. Manuals do not provide useful technical information. Most details concerning computers were assumed to be known by a potential user of QNX.

The C language

Knowledge about the syntax was obtained through literature. Second, TURBO C was installed on a DOS operated PC that provides an excellent tutorial to introduce programming in C. Advanced issues were tackled after the basics were understood. Many small programs were written and tested. A great deal of efforts was spent on developing modular programs including global variables.

Problem areas

Operations on variables of type pointer (variable addresses) were identified to be the most difficult topic in C. This requires knowledge of the organisation of memory in a PC and protection mechanisms of certain address spaces. Bitwise operations required deep knowledge of binary algebra.

The PC control system

Observation of the current proceedings and research meetings led to a brief understanding of what was sought to be accomplished. An illustration of the many tasks to be performed on the PC was very helpful to estimate the complexity of the system where the interface has to fit in. The principle foundations of digital process control was obtained through course work and studies in the field of process control theory.

Problem areas

It was difficult to state in great detail what features the user interface has to provide for

the control system. The anticipation of future requirements was challenging since too many unknowns were identified. Many present developments are considered provisory and are apt to fundamental changes in the future.

(ii) PHASE II - Definition of interface specifications

The primary interface specifications could be identified once the design objectives for the control system were understood. In addition, the knowledge of commercial software and the menu mechanisms led to the intention to incorporate similar features into the interface.

Problem areas

It was more complicated to find an optimal modular structure of the interface modules. The specification of hidden interface features affecting coding, compiling (the choice of memory models for modules), and linking required the development of a number of preliminary versions that were tested and analyzed. A great deal of alternatives can be conceived regarding the choice of memory models and the number or tasks that make up the interface program. It seems to be not obvious which alternative to consider best suited.

(iii) PHASE III - programming and iterative revisions

Since the evident and hidden specifications were known at this point of the interface development, coding unfolded to be straightforward in the beginning. Initial coding was rather directed to the definition of logical branching including than to the perfection of displays and menus. Repeated improvements of the display and menu features led to substantial improvements step by step. Complications arose when it was decided to modify the structure of the intermediate version of the program. Some of the modules present in the final version were subsequently added which required often elaborate rearrangements of the code. Errors were thus introduced that had to be identified and corrected. This was more challenging the bigger the volume of code had grown.

Problem areas

To understand the operations and options of the compiler and linker proved to be more problematic than to understand the peculiars of the language C. Too many options and not transparent documentation led to a decision to apply trial-and-error. Through conducting compiling and linking experiments issues would clarify and the solution sought could be implemented. Especially QNX specific functions for task creation and inter-task communication required a strong amount of efforts since programming examples were scarce in the documentations. Some hard-headed errors led to contacting the technical service of Quantum Software. Verbal consultations proved to be only partly helpful.

(iv) PHASE IV - Final testing and inclusion into the overall system

This step involved the linking of the workable version of the user interface to the network of tasks related to data acquisition and process control. This was done by installing the interface files on the harddisk of the PC to host the system. Since QNX maintains a UNIX style file system with owner and user rights, care had to be taken to change the file flags accordingly. When logged into the system as super-user this was a straightforward operation. In addition, the directory specifications in the user interface files had to be checked to match the assumptions made by the control and data acquisition tasks. Execution of the interface task and the activation of the above task network initially caused corruption due to a confusion of memory addresses or a corrupt logical sequence of task creation and communication primitives.

Problem areas

Since the hardware components on the machine were not completely installed to test the machine, missing activities on the screen could not easily be identified as coding errors. Non-functioning hardware could have also been a probable cause. The synchronisation of all tasks required a trial-and-error approach based on incidence since the complexity of the overall system is not transparent. It was not possible to test the interface program without the specialist in charge for the control task network and vice versa. This might prove a bottleneck if independent steps are to be taken to develop both the interface and the control task network further.

8.2 Recommendations

At the present stage of the project it is more than likely that fine-tuning of the program hierarchy has to be performed. This might require a change of the order of appearance of the task creation macros in the module 'fw_do.c'. Before modifications are tackled, it ought to be very clear what effect this is supposed to achieve. To avoid memory shifting and subsequent corruption, it must be made sure that an alteration does not assign values of a global variable to a local one. If so, the variable affected should be declared within the main module 'if_start.c' to join the group of other global variables. As for debugging, the virtual-screen technique described in section 7. should be applied.

In case CPU utilisation of the user interface related task turns out to inhibit proper operation of the data acquisition and control algorithm tasks first an adjustment of the scheduling policy could be made. Best suited is a technique that involves explicitly reducing the priorities of the interface tasks while explicitly boosting the priorities of the control system and data acquisition related tasks. A second approach could rely on switching of scheduling to round-robin. The equally distributed allocation of CPU-time slices among all tasks might cause an improvement. It is likely that the interface task as the parent of all tasks is given to much CPU time when default scheduling is applied.

In the worst case a second machine can be used for display purposes. QNX 4.0 is designed to maintain a PC network. In that event one should also consider the purchase of QNX WINDOWS which is supposed to provide real-time graphics display. The structure of the current interface could serve as a pattern to logically implement a similar interface on the

second machine, although display quality could be a great deal improved. Icons such as clocks and scales can be included which offers an excellent means to replace the status display task which was considered as a CPU-time-saving expedient.

If real-time printing of sensor readings is mandatory there might be no alternative other than to use a second PC for the display. Too many updated parameters will have to be printed which limits CPU allocation to the control algorithms and the data acquisition tasks.

The current version of the interface could have minor enhancements, e.g. units of measure ought to be displayed they are decided.

7. References

Andrews, G.R., and Schneider, F.B.
Concepts and notations for concurrent programming
Computing Surveys, 15, 3-43 (1983)

Denning, R.C.
The working set model for program behaviour
Comm. ACM, 11, 323-33 (1968)

Habermann, A.N.
Prevention of system deadlock
Comm. ACM, 12, 373-7 (1969)

Lister, A.M., and Eager, R.D.
Fundamentals of Operating System, 4th Edition,
MacMillan Eduacation Ltd. (1988)

Mellichamp, Duncan A.
Real-Time Computing
With Applications to Data Acquisition and Control
VNR-Van Nostrand Reinhold Company (1983)

Peterson, James L., and Silberschatz, Abraham
Operating System Concepts, 2nd Edition
Addison-Wesley Publishing Company (1985)

Reiling, A.
PDA-MDA-DNC: An overall concept for injection moulding
German Plastics 79 (1989)1, pp. 29/33

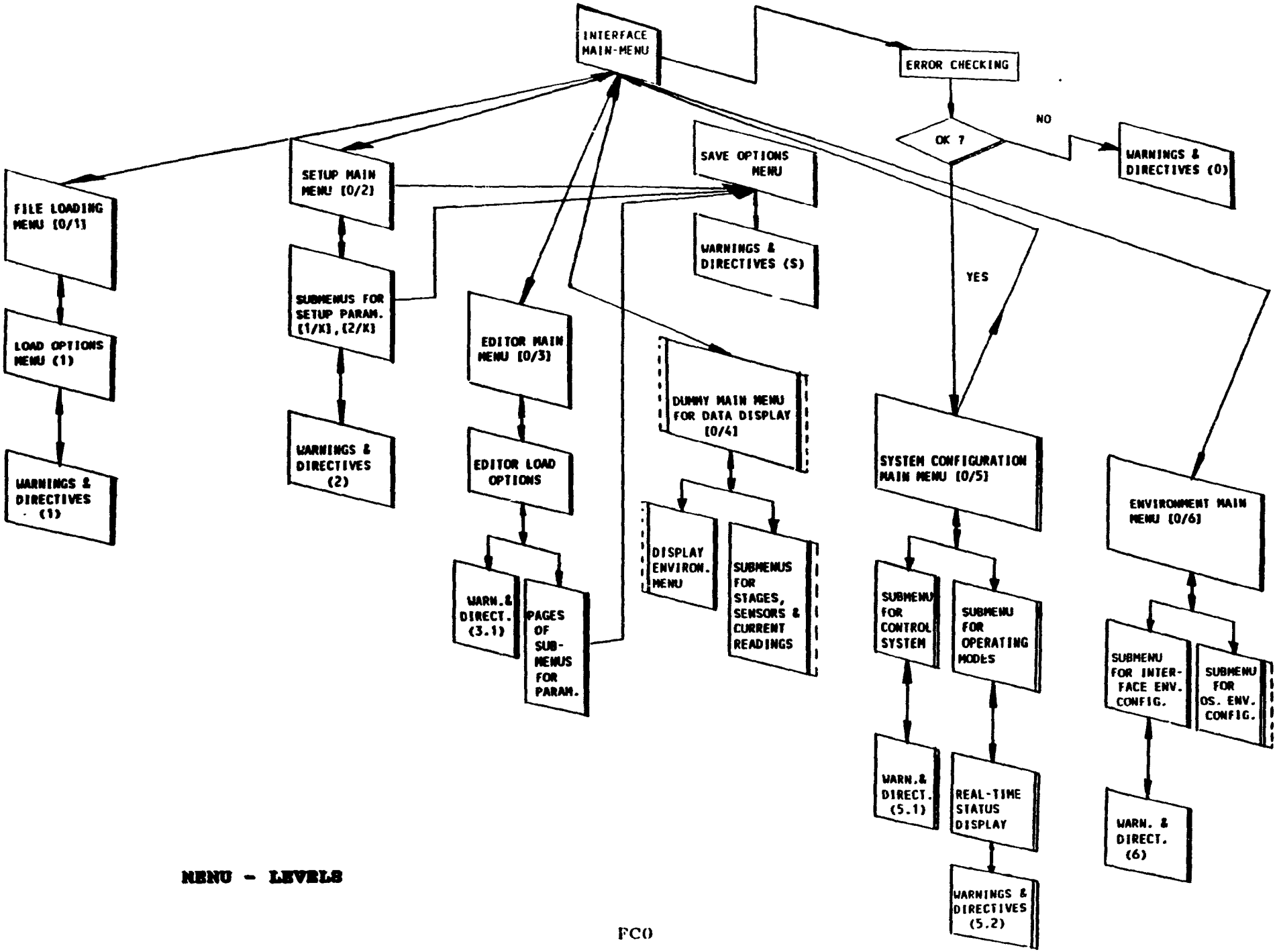
Schwab, E.
Agreement on Uniform Interfaces to the Master Computer
German Plastics 79 (1989)11, pp. 1133/1134

Tokheim, Roger L.
Theory and Problems of Microprocessor Fundamentals
Schaum's Outline Series
McGraw-Hill Book Company (1983)

10. APPENDIX

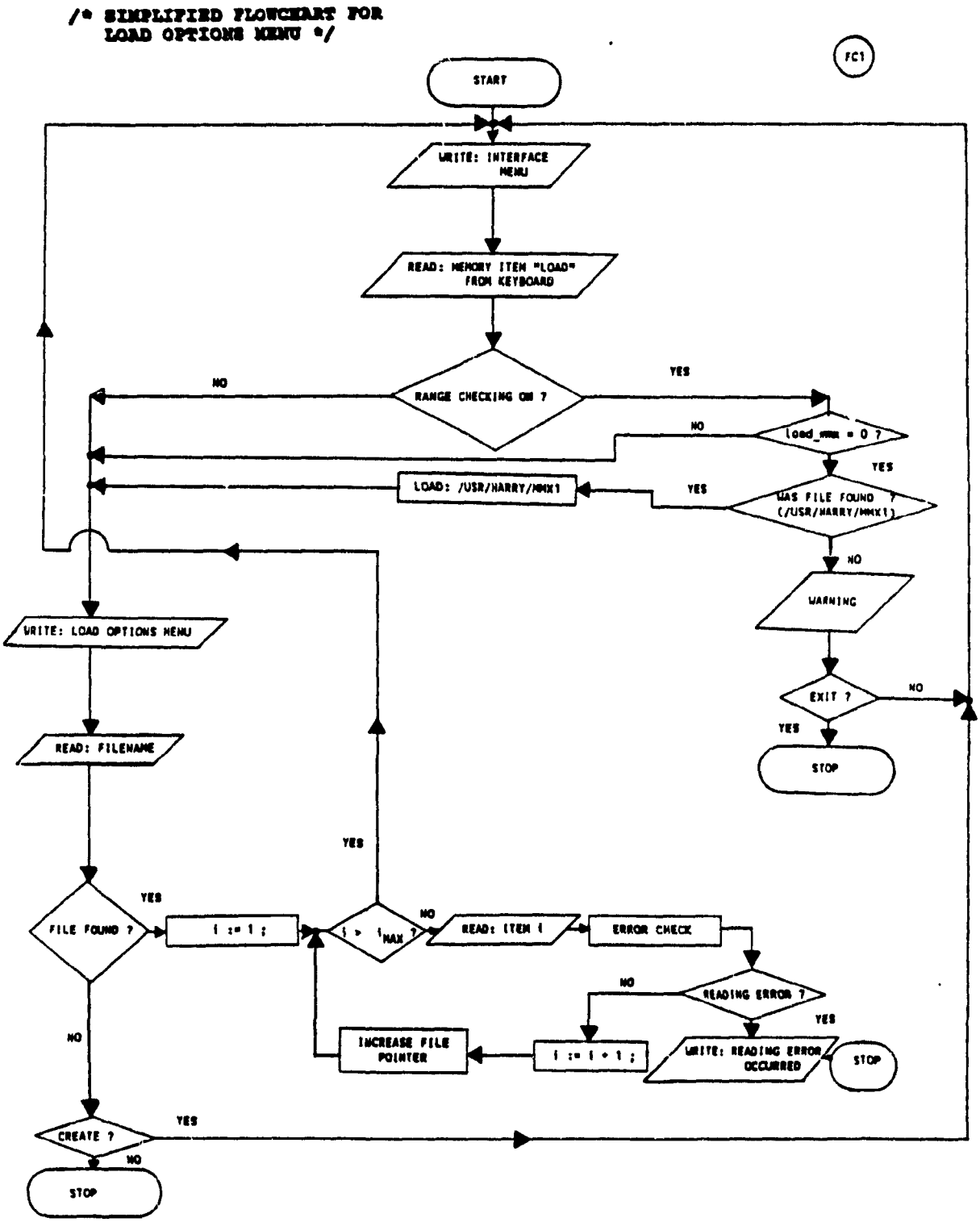
10.1 Illustration of menu levels

10.2 Flowcharts


INTERFACE MAIN-MENU
ERROR CHECKING
OK ?
NO
YES
WARNINGS & DIRECTIVES (0)
FILE LOADING MENU (0/1)
LOAD OPTIONS MENU (1)
WARNINGS & DIRECTIVES (1)
SETUP MAIN MENU (0/2)
SUBMENUS FOR SETUP PARAM. (1/X), (2/X)
WARNINGS & DIRECTIVES (2)
SAVE OPTIONS MENU
WARNINGS & DIRECTIVES (S)
EDITOR MAIN MENU (0/3)
EDITOR LOAD OPTIONS
WARN.& DIRECT. (3.1)
PAGES OF SUB-MENUS FOR PARAM.
DUMMY MAIN MENU FOR DATA DISPLAY (0/4)
DISPLAY ENVIRON. MENU
SUBMENUS FOR STAGES, SENSORS & CURRENT READINGS
SYSTEM CONFIGURATION MAIN MENU (0/5)
SUBMENU FOR CONTROL SYSTEM
SUBMENU FOR OPERATING MODES
WARN.& DIRECT. (5.1)
REAL-TIME STATUS DISPLAY
WARNINGS & DIRECTIVES (5.2)
ENVIRONMENT MAIN MENU (0/6)
SUBMENU FOR INTER-FACE ENV. CONFIG.
SUBMENU FOR OS. ENV. CONFIG.
WARN. & DIRECT. (6)

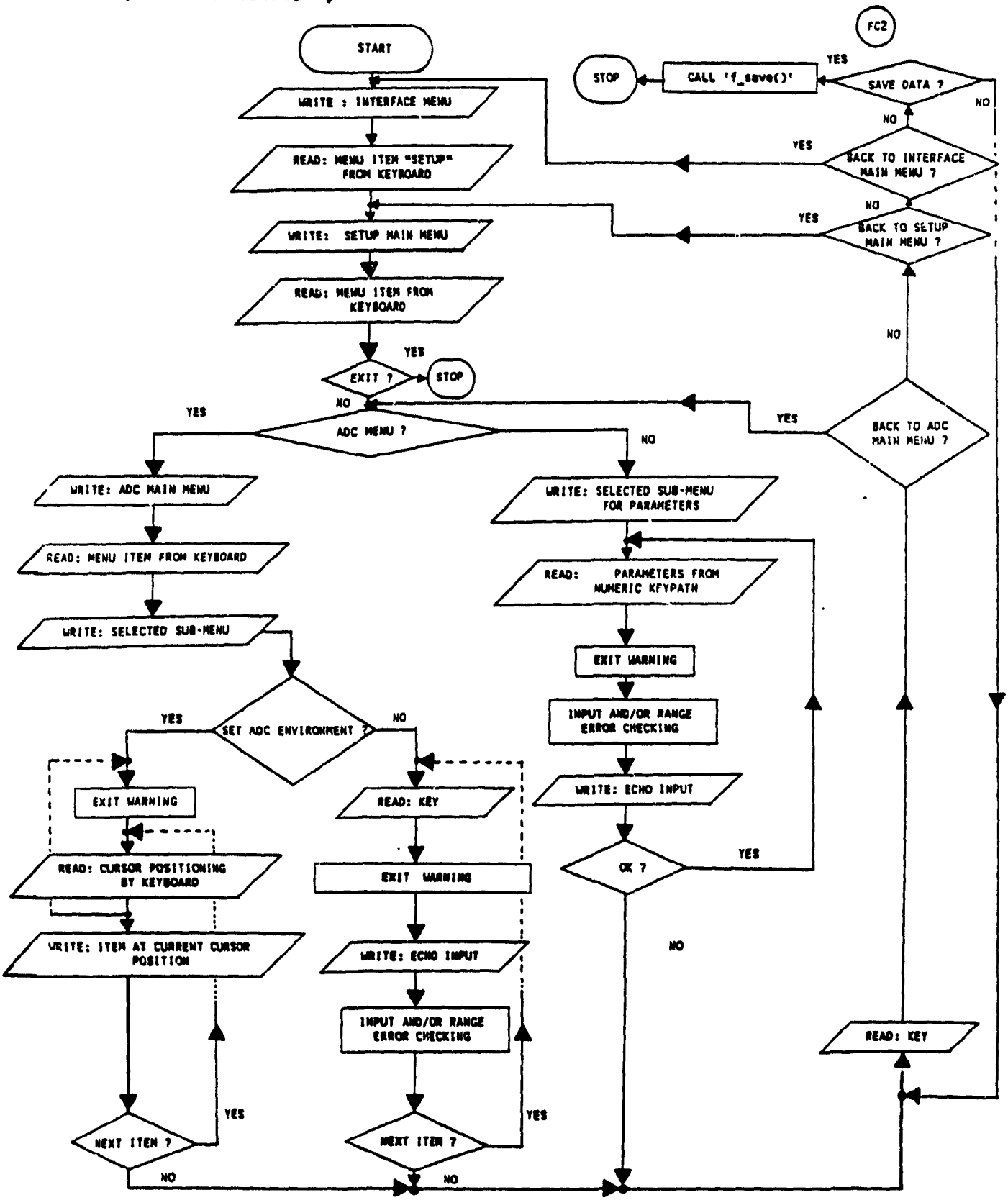
MENU - LEVELS

Flowchart 1


/* SIMPLIFIED FLOWCHART FOR
LOAD OPTIONS MENU */
FC1
START
WRITE: INTERFACE MENU
READ: MEMORY ITEM "LOAD" FROM KEYBOARD
RANGE CHECKING ON ?
YES
NO
load_mm = 0 ?
NO
YES
WAS FILE FOUND ? (/USR/HARRY/MMX1)
YES
LOAD: /USR/HARRY/MMX1
NO
WARNING
EXIT ?
NO
YES
STOP
WRITE: LOAD OPTIONS MENU
READ: FILENAME
FILE FOUND ?
YES
i := 1 ;
i > i MAX ?
NO
YES
READ: ITEM i
ERROR CHECK
READING ERROR ?
NO
YES
i := i + 1 ;
INCREASE FILE POINTER
WRITE: READING ERROR OCCURRED
STOP
NO
CREATE ?
YES
NO
STOP

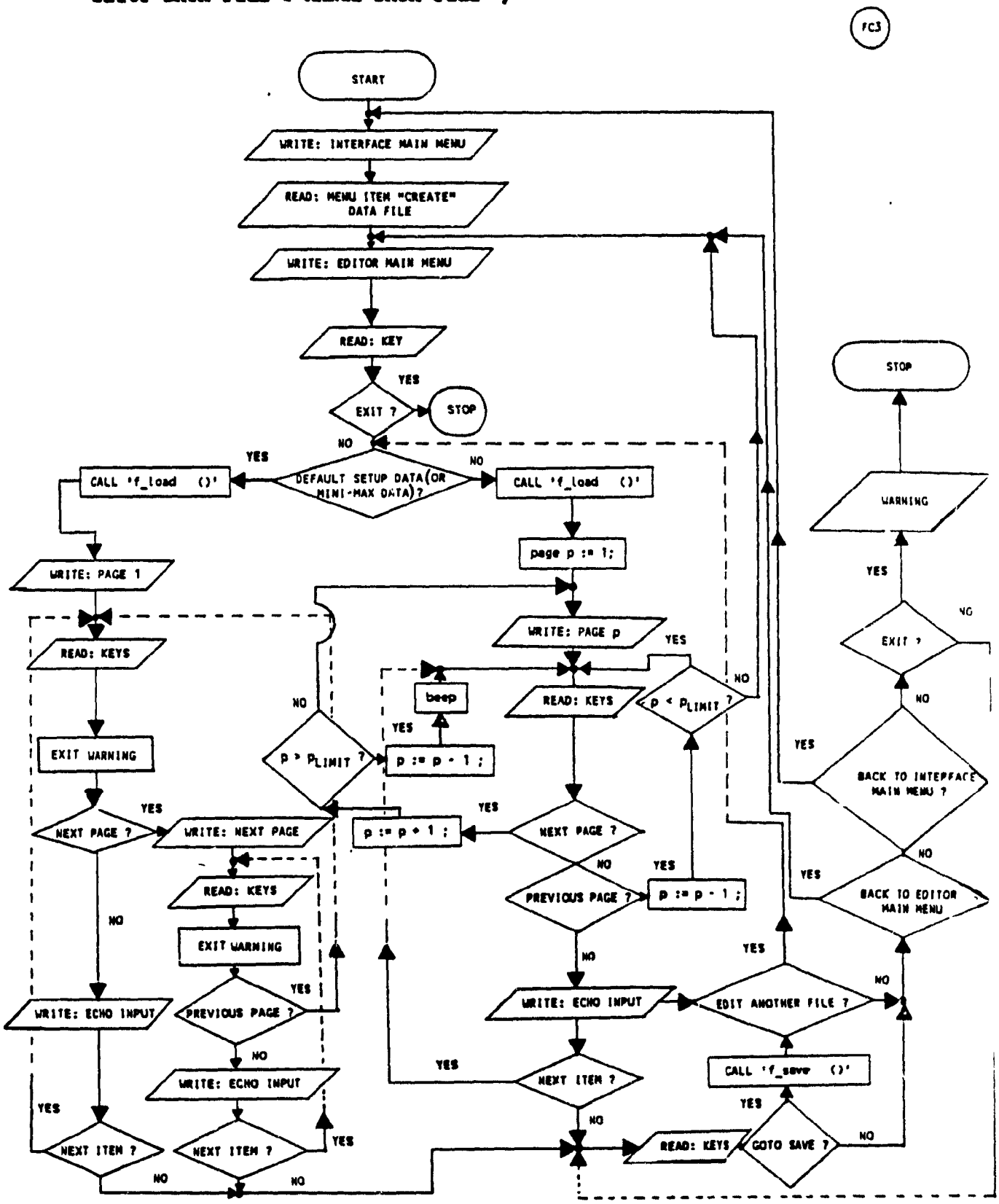
Flowchart 2

/* SIMPLIFIED FLOWCHART FOR SETUP PARAMETER MENU
(ADC-PARAMETERS) */



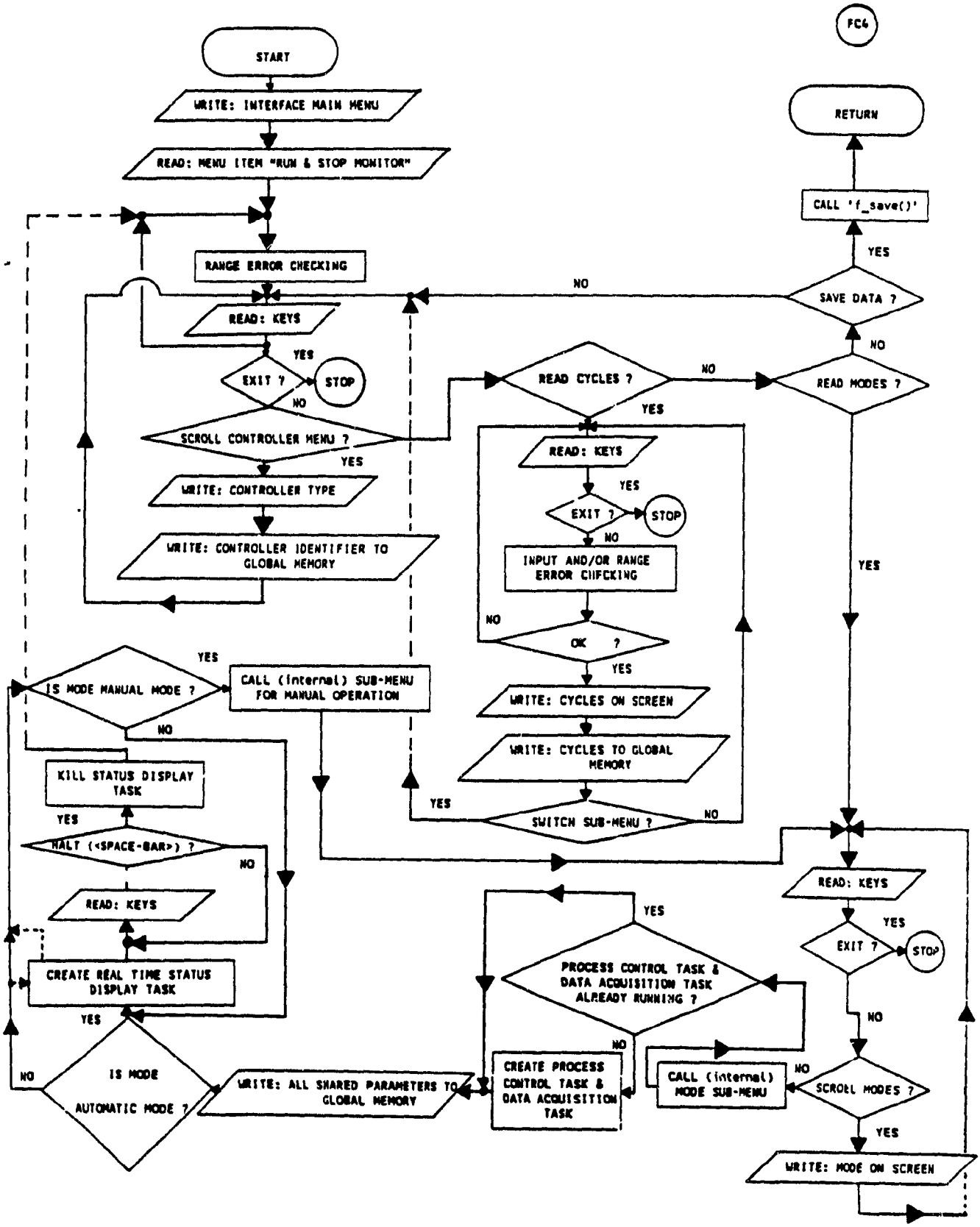
Flowchart 3

/* SIMPLIFIED FLOWCHART FOR EDITOR FOR
SETUP DATA FILE & RANGE DATA FILE */



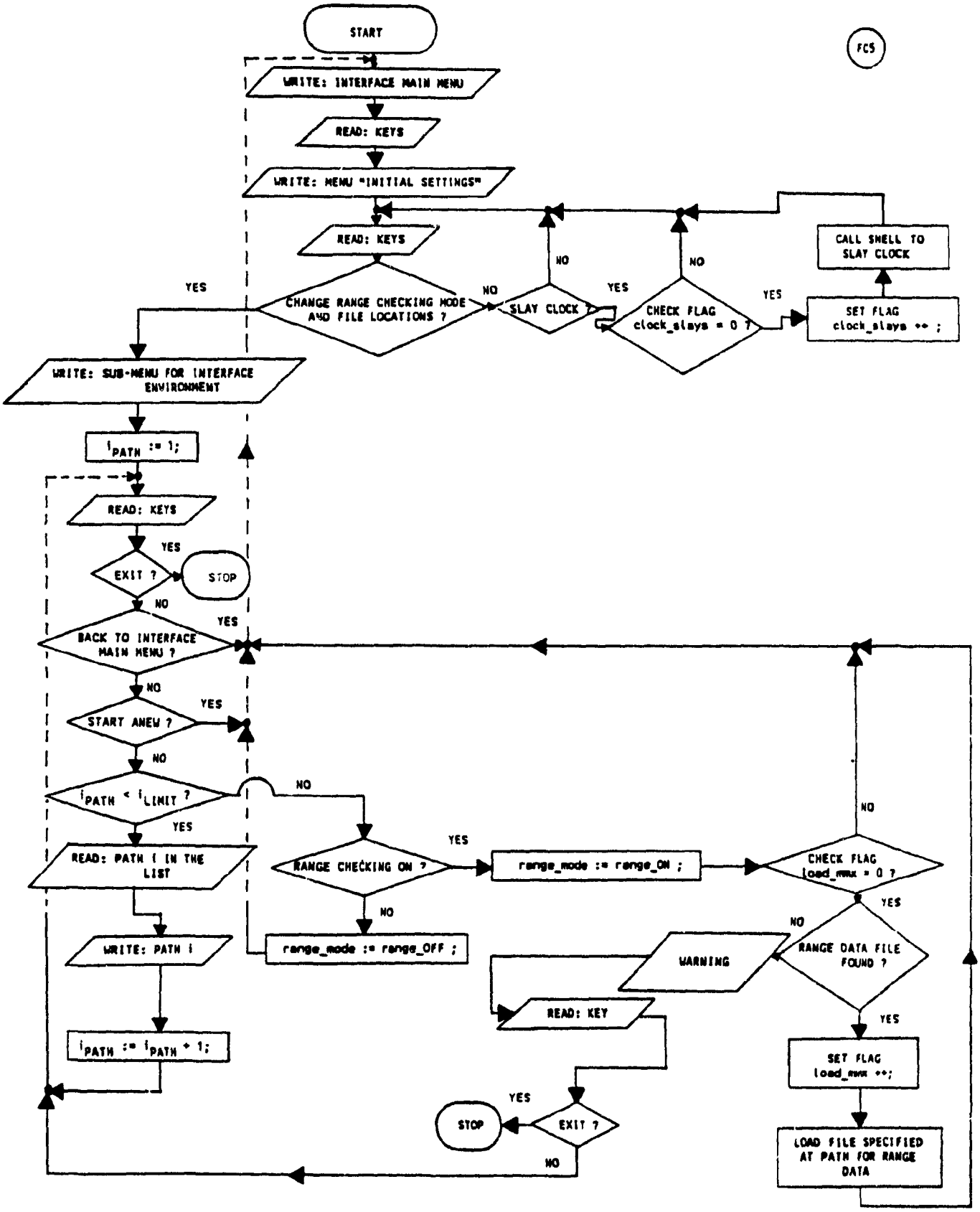
Flowchart 4

/* SIMPLIFIED FLOWCHART FOR CONTROL SYSTEM CONFIGURATION MENU */

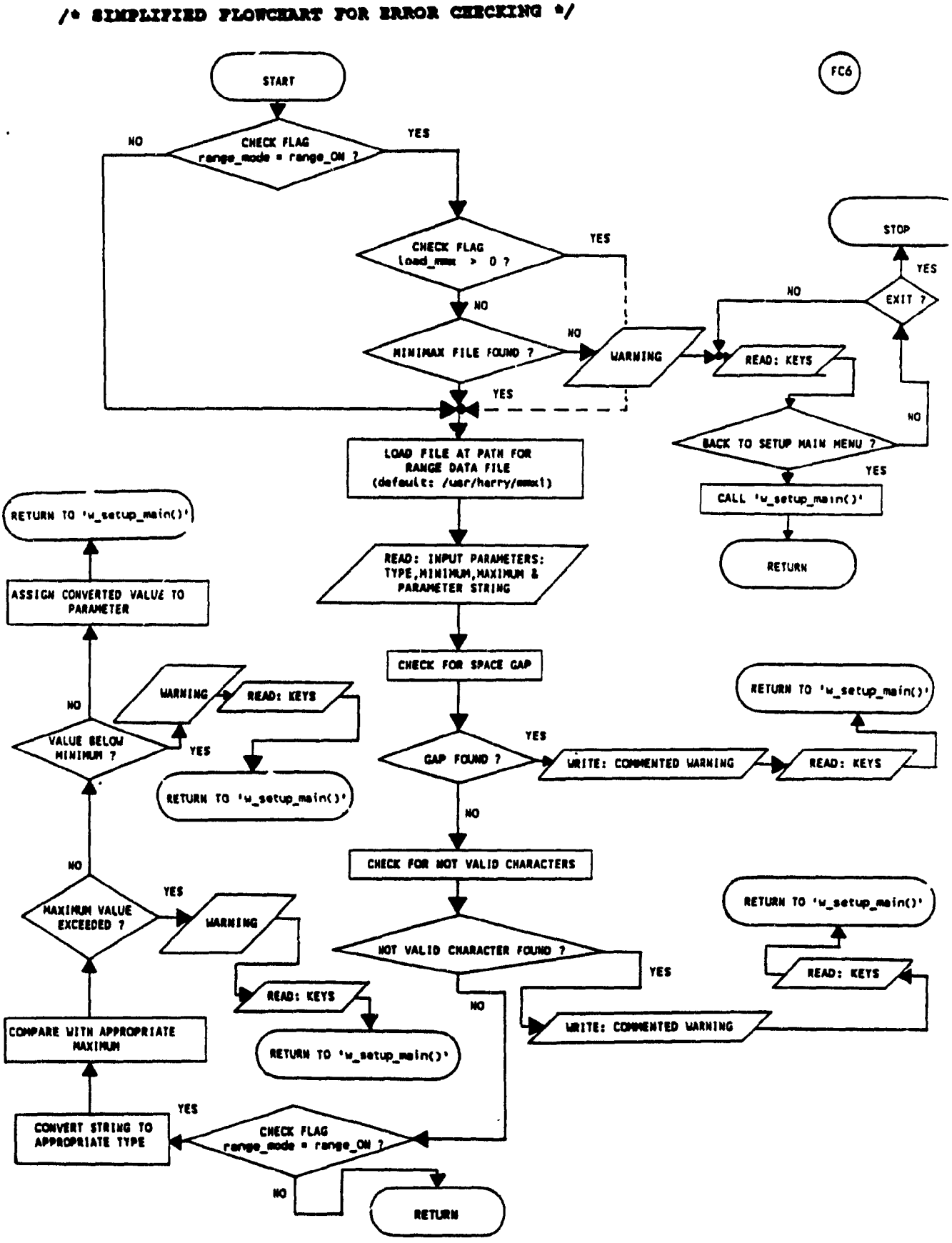


Flowchart 5

/* SIMPLIFIED FLOWCHART FOR INTERFACE ENVIRONMENT MENU */

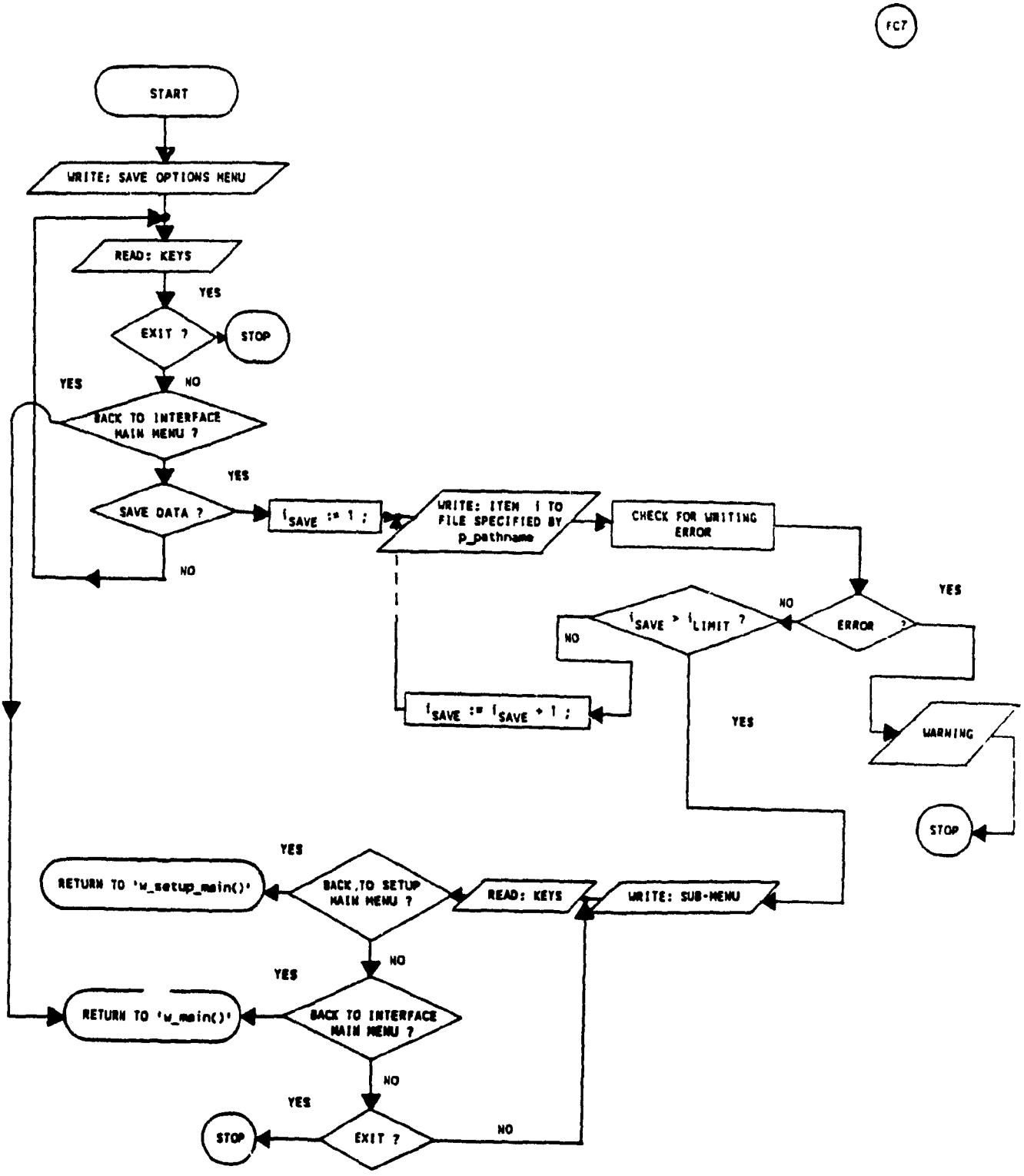


Flowchart 6


/* SIMPLIFIED FLOWCHART FOR ERROR CHECKING */
START
FC6
CHECK FLAG range_mode = range_ON ?
NO
YES
CHECK FLAG load_mmx > 0 ?
YES
NO
MINIMAX FILE FOUND ?
NO
WARNING
READ: KEYS
YES
STOP
YES
EXIT ?
NO
BACK TO SETUP MAIN MENU ?
NO
YES
CALL 'w_setup_main()'
RETURN
LOAD FILE AT PATH FOR RANGE DATA FILE (default: /usr/harry/mmx1)
READ: INPUT PARAMETERS: TYPE,MINIMUM,MAXIMUM & PARAMETER STRING
CHECK FOR SPACE GAP
GAP FOUND ?
YES
WRITE: COMMENTED WARNING
READ: KEYS
RETURN TO 'w_setup_main()'
NO
CHECK FOR NOT VALID CHARACTERS
NOT VALID CHARACTER FOUND ?
YES
WRITE: COMMENTED WARNING
READ: KEYS
RETURN TO 'w_setup_main()'
NO
CHECK FLAG range_mode = range_ON ?
YES
NO
RETURN
CONVERT STRING TO APPROPRIATE TYPE
COMPARE WITH APPROPRIATE MAXIMUM
MAXIMUM VALUE EXCEEDED ?
YES
WARNING
READ: KEYS
RETURN TO 'w_setup_main()'
NO
VALUE BELOW MINIMUM ?
YES
WARNING
READ: KEYS
RETURN TO 'w_setup_main()'
NO
ASSIGN CONVERTED VALUE TO PARAMETER
RETURN TO 'w_setup_main()'

Flowchart 7

/* SIMPLIFIED FLOWCHART FOR SAVING */



NOTE

The procedure for making new executable files after modifications were made to the source code are as follows:

(1) Rename the modules '*.c'. The module names and the function declarations in the header files 'graphs.h' and 'f_decl.h' must match precisely. This affects expanding the names of the files converted to DOS format (ASCII). The characters: 'f', 'w', and 'fw' have to be separated from the remaining character string by a short case character, '_'.

For example, the file 'fdecl.h' found on diskette 1 must be renamed to yield 'f_decl.h'. The filename 'btemp' of a file found on diskette 2 must be expanded to yield 'barreltemp' and so on.

Use the QNX functions 'more' or 'less' to view the contents of the above header files and the appropriate source code file ("*.c") for the correct names.

(2) For application of the WATCOM compiler and linker on QNX, the files "*.c" and "*.h" found on diskette 1 must be converted to QNX 4.0 specific format. Use the QNX function 'textto -l -z' for the job.