AN ABSTRACT OF THE THESIS OF

Hans-Werner Schaal for the degree of Master of Science in Electrical and Computer Engineering presented on October 10, 1989

Title: TASKMASTER II, A Versatile Distributed System for Control

Abstract approved: Redacted for Privacy

James H. Herzog

This thesis discusses a special type of a versatile, distributed control system, the Taskmaster II. It includes a detailed specification of a readily realizable Taskmaster system utilizing a layered communication among units, oriented on the ISO OSI seven layer model. The specification was developed from the analysis of previous research conducted at the Department of Electrical and Computer Engineering at Oregon State University and a survey of modern state-of-the-art hardware environments. The Taskmaster is an open system and embodies hardware and software components.

Based on the developed specification, a functional analysis for the implementation of a revised Taskmaster operating system on distributed hardware environments is presented. It is intended to serve as a guideline for the porting of the Taskmaster Operating system to hardware components of any microprocessor.

Utilizing this analysis, the core of the Taskmaster Operating system was implemented on a new hardware environment, satisfying several system requirements. One particular condition was to keep components of the new

Taskmaster system compatible to those of the old system. This was realized by preserving the lower communication layers and a particular communication protocol. In this implementation, a Motorola MC68C11 microcontroller and a Motorola evaluation board were selected and successfully employed as hardware nodes of the distributed Taskmaster System.

The performance of the implementation was tested and discussed. It could be shown that the new system is a truly open system and able to employ hardware units of different vendors, utilizing the enhanced computational capabilities of recently available hardware components.

Taskmaster II,
A Versatile Distributed System for Control

by

Hans-Werner Schaal

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed October 10, 1989

Commencement June 1990

APPROVED:

Redacted for Privacy

Associate Professor, Electrical and Computer Engineering
in charge of major

Redacted for Privacy

Head of Department of Electrical and Computer Engineering

Redacted for Privacy

Dean of Graduate School

Date thesis is presented October 10, 1989

Typed by Hans-Werner Schaal for Hans-Werner Schaal

Acknowledgement

I wish to express my thanks to my major professor and thesis advisor Dr. James H. Herzog for his encouragement and guidance throughout this study.

I would also like to thank Motorola Inc. for the donation of the 68HC11 Evaluation Board.

Last, but not least, I would like to thank my fellow students Yong Thye and Andreas Weisshaar for their frequent advice and encouragement throughout my studies.

LIST OF ABBREVIATIONS

		Page
A/D	Analog-to-Digital	30
ACP	Assembled Command Packet	45
ASCII	American Standard Code for Information Interchange	4
COLAN	Local Area Network for Control Use	5
CPU	Central Processing Unit	14
EITA	Execute an Immediate Task!	25
EQTA	Execute a Queued Task!	25
EVB	Evaluation Board	35
FFE	For Future Extension	41
FIFO	First-in-First-Out	21
HCMOS	High-Density Complementary Metal-Oxide Semiconductor	30
HCP	Host Command Packet	18
I/O	Input/Output	28
ISO	International Standards Organization	2
LAN	Local Area Network	5
LSB	Least Significant Bit	51
MCU	Microcontroller Unit	27
NE	Normal Extended	32
NIU	Network Interface Unit	5
NSC	Normal Single Chip	32
OSI	Reference Model of Open Systems Interconnection	2

LIST OF ABBREVIATIONS cont.

PC	Personal Computer	3
SBC	Single Board Computer	14
SBCU	Single Board Computer Unit	27
SCU	Single-Chip Units	27
SS	System Scheduler	3
SSIR	Service the Serial Interrupt!	24
SSW1	System Statusword 1	46
SSW2	System Statusword 2	46
TIC	Timer Input Compare	31
TOC	Timer Output Compare	31
TOS	Taskprocessor Operating System	3
TPU	Taskprocessing Unit	2
TSW	Task Statusword	45

TABLE OF CONTENTS

TABLE OF CONTENTS cont.

LIST OF FIGURES

LIST OF TABLES

PREFACE

The idea of the Taskmaster system as a specialized example of a modern distributed control system has been developed at Oregon State University during the last five years based on design methodologies in real-time control [1-7]. This work tries to briefly summarize this development and to give a general guideline for the implementation of a revised Taskmaster system on modern hardware environments. In addition, the core of this revised Taskmaster was implemented based on selected hardware components.

Chapter 1 gives a general overview on the Taskmaster system and discusses applications and previous research. Chapter 2 is a detailed analysis of the revised Taskmaster developed in this work. It specifies the desired properties with respect to currently available hardware components and gives a guideline for their implementation. It is intended as an aid to efficiently port the revised Taskmaster system to any appropriate hardware environment.

Chapter 3 describes the implementation of the revised Taskmaster system on a selected, modern hardware environment. The performance of the system is described and discussed in Chapter 4. Finally, Chapter 5 gives suggestions for future research projects.

TASKMASTER II, A Versatile Distributed System for Control

CHAPTER 1

1. INTRODUCTION

1.1. Overview

A natural evolution of real-time control demanded system designs which employ several high performance microprocessors and coprocessors, each specialized for particular functions. Distributed systems are arrangements of a variety of computing facilities which are established at separate locations but are still able to function in a cooperative manner [4-6]. These units host the computing facilities of a distributed system and perform communication functions. They are often referred to as nodes. A general distributed system is depicted in Figure 1-1.

In distributed control systems, complex control activities are decomposed into several control assignments and tailored to be performed by particular units [7]. The resulting modularity implies the possibility of highly specialized implementations of these units, which is frequently desired in control applications. Modularity also usually simplifies maintenance and troubleshooting.

Recently, a tendency has evolved to keep distributed systems open [6]. An open distributed system allows the utilization of components supplied by different vendors. This usually simplifies the optimization of particular units and often reduces system costs. Standards have been developed for the implementation of open systems, one of which is the Reference Model of Open Systems Interconnection (OSI) which was developed by the International Standards Organization (ISO). It is often referred to as the ISO OSI Reference Model and will be discussed later.

These standards are guidelines for establishing and conducting communication among units of distributed systems. The communication is necessary to render coordinated activity of the system's units and is usually a crucial issue in the system's performance. The ISO OSI Reference Model arranges the functions and rules of communication among units into a hierarchy of subsets, called layers. The lowest subset in this hierarchy is the Physical Layer which describes the physical connection among units [6,7].

One example of a distributed control system, the Taskmaster, has been developed at the department of Electrical and Computer Engineering [2,3]. As depicted in Figure 1-2, the Taskmaster System consists physically of several hardware units, called the Taskprocessor Units (TPUs), a communication network to provide physical connection, and at least one system scheduler (SS). The SS is usually implemented as a Personal Computer (PC) and provides an intelligent interface between a human operator and the network. Since the SS is connected to the network it can communicate with specified TPUs utilizing a special command format. Each TPU itself interfaces to the network and to possible applications. It provides resources for the control of applications and for communication through the network.

Each TPU hosts a sophisticated Taskmaster operating system (TOS) which enables the unit to perform in a semi autonomous manner. If no commands are issued to the network, each TPU performs autonomously without interaction with the SS or any other unit. The schedule describing the desired activity is installed in software form in each TPU. Control of an application by a TPU can be referred to as background activity.

In this configuration, the TPU resembles an independent controller unit rather than an element of a larger distributed system. Even when currently performing according to the background schedule, the TPU is still able to receive commands from the communication network in a foreground activity. As depicted in Figure 1-2, background activity is an interaction between the TPU and its application. The foreground activity is the interaction between the TPU and the communication network.

Special foreground commands are defined to control the background activities. Highly efficient control of the system's TPUs can be achieved by issuing a sequence of particular foreground commands [3]. A human operator can utilize these foreground commands to directly control the TPUs. In a minimum configuration, the SS needs only to be implemented as a dumb terminal. It is used to inject foreground commands into the network.

If a SS with considerable computing power is available, like a PC, a control program in a high level language is suitable to control TPUs if it is designed to issue foreground commands as output variables. Each TPU is able to perform all application services like an independent microprocessor based unit and in addition interfaces this assembly level control to a high level language. The only requirement for the high level language is to be able to send and to receive characters compatible to the American Standard Code for Information Interchange (ASCII). With a fairly limited set of foreground commands, complex control and monitoring activities can be performed. It is interesting to note that the effective application bandwidth of this setup is considerably higher than the bandwidth requirements of the communication network [3].

One particular property of the TOS is that in addition to well-known software structures, like subroutines, it also employs a task structure which was specifically developed for the Taskmaster system. A task is an independently scheduled event which requires and services system resources [3]. In other words, a task is a piece of code whose execution can be scheduled in a particular unit by issuing a distinctive command to the network.

1.2. Applications and Previous Research Efforts

Distributed systems like the Taskmaster find a wide area of applications. Examples are Local Area Networks (LANs) and LANs for control use (COLANs) [4,5,8-10]. In LAN applications the TPU is employed as a network node and its main occupation is to ensure proper reception and forwarding of data packages to a Host as depicted in Figure 1-3. In this configuration, each unit comprises a Host, a TPU, and possibly a network interface unit (NIU).

A typical control oriented application is the coordinated activity of several stepper motors in a complex positioning mechanism like a robot arm. Each stepper motor is controlled by its own TPU. The proper addressing and scheduling of selected tasks allows the coordinated action of the TPUs and their applications.

Another, more sophisticated example is the control of an assembly line as depicted in Figure 1-4. Usually, an assembly process involves the sequential processing of a production unit through several stages. To control this environment, one TPU is designated as a Master and is interfaced to a Host Computer. This is usually a dedicated personal computer or mini computer. Each stage of the assembly process has an associated machine which is controlled by a TPU.

A preprogrammed sequence of background commands to control the associated machine is present in the TOS of each TPU. Commands can be issued from the host node to influence the performance of selected TPUs by modifying their background activity. This as a consequence regulates the action of the associated machine. The system could be highly efficient if stage x sends a command to its successor, stage x+1, upon completion of the manufacturing step of its corresponding machine. This allows an optimized takeover of the processing by the successor machine (see Figure 1-4).

One objective of previous research was the implementation of LANs and COLANs. LANs find a wide application in modern offices where they interconnect several Hosts (usually PCs). They provide resource sharing of expensive equipment (e.g. printers) and special services (e.g. mail and messages, similar to a UNIX environment). Educational LANs are centralized LANs and usually include a Master node (the teacher) which connects to several slave nodes (the students) [12-13].

Initially, in 1984, emphasis was placed on applications which showed the feasibility of the Taskmaster System [2-3]. Some experimental LANs were designed [14-16]. Then some efforts were made to implement the idea of an open system in the Taskmaster [17]. For the physical layer in the communication network, a bus structure (RS-485) was used.

The operating system was first implemented on an Intel 8048 CPU and then ported to the Intel 8051. Due to upward compatibility, the core of the operating system was only slightly modified. This simplified the porting but did not fully exploit the enhanced computing power of the new processor. Also, applications demanded utilization of TPUs supplied by different microprocessor vendors. The above encouraged a fresh approach on a new hardware environment.

CHAPTER 2

2. FUNCTIONAL ANALYSIS OF THE TASKMASTER

In the following Chapter, a detailed conclusion of considerations necessary for the implementation of a TOS is presented. It is a general, hardware independent reasoning, intended to serve as a guideline for the implementation of a TOS on any given hardware environment. In the first section, the specifications of the system are discussed. In Section 2.2., conclusions are drawn from these specifications. As a result, a general state description of the system is presented. Section 2.3. describes some available design options in the implementation of the system.

2.1. Specifications of the Taskmaster System

The Taskmaster system consists of both hardware and software. Physically it can be described as a distributed system which comprises several nodes (see Figure 1-2). These nodes are implemented as hardware units, called TPUs and interface to a common communication network. TPUs are usually similar to each other in morphology (see Figure 2-1). Each node hosts a special operating system, the TOS which has sufficient intelligence to control applications of the TPUs and to communicate to the network through an interface.

Another hardware element of the Taskmaster system is the SS. It has two functions. First, it serves as an interface between a human operator and the network. This enables the operator to control background activities of specified TPUs by issuing designated commands to the network. Second, it can run a high level language program which issues commands to the network autonomously. The SS is interfaced to the network, either directly or through a TPU.

Organizational requirements of the system include autonomy, communication, automatic recalling of services, and some means of interruption of services in each unit.

2.1.1. <u>Autonomy of each Unit</u>

Each TPU should be implemented as an autonomous subsystem that is able to function independently without connection to the network. Thus, if there is no action on the network, each TPU concentrates on servicing applications as a background activity. Thus, the TOS, which is present in each TPU, needs to be able to organize this background activity independently. The TOS incorporates dedicated pieces of software and data to assure the desired control activity.

The TPU needs to be capable of performing the same control application services that an independently working microcontroller system could provide. The TPU shields the application environment from the communication network. Thus, the application environment cannot detect whether it is controlled by a TPU (which is connected to a communication network) or a by an independent system.

2.1.2. <u>Communication among Units</u>

Besides performing application services as background activity, each TPU also needs the ability to communicate with the SS and possibly with other network nodes. This requires the execution of a foreground activity that has priority over the background activity and is exclusively devoted to serve communication to the SS and other nodes. The priority needs to be higher since each node must be able to receive foreground commands issued by the host or possibly other units at any time, regardless of the current background activity. This is necessary to entitle foreground commands to direct background activities.

If multiprocessing is available in the TPU, then foreground and background activities can be realized as separate processes, provided a clear line of authority is established. If only single processing capabilities are available, the desired line of authority can be implemented by utilizing interrupts.

2.1.3. Command Accumulation and Automatic Recalling of Services

Due to their complexity, it is often more efficient to describe application services by a sequence of foreground commands rather than by a single command. This, however, requires collection and administration of commands which were received previously by TPUs from the network. A TPU should not be required to execute services immediately upon reception of respective commands. Instead, it should be able to accumulate several commands and to delay their execution. This entitles each TPU to initiate background mode operation at a specific time and to direct it according to a batch of commands.

In addition, many control applications are of a recursive nature and require the repetition of particular activities by control units. An example is a particular stage of an assembly process in a factory environment. In this stage, a particular sequence of actions is performed, say three holes are drilled in a cylinder head block. The factory might produce 1000 engines a day, which requires the repetition of the assignment "drill three holes" 1000 times a day.

All actions necessary to accomplish the drilling of the three holes can be viewed as an application service. It is then desired to recall this service automatically upon the arrival of each new cylinder head block at the discussed stage. This requires the storing of a sequence of auxiliary actions in a special buffer located on the TPU. Processed sequentially, these auxiliary actions perform the application service.

In a predefined operating mode, the TPU can then provide repetitive services by recursively accessing the buffer. In the Taskmaster System, this operating mode can be initiated by a specific command (by the SS or another TPU). This command will be received through the network by the appropriate TPU.

2.1.4. Interruption of Services upon Conditions

The control of a complex system by the taskmaster usually follows a typical scenario. The whole control assignment is distributed over several TPUs. Each TPU is optimized for a particular subassignment in hardware and software. Each TPU executes its background activity independently. At times, the background activity is postponed to conduct communication as a foreground activity. However, operating time in the foreground mode is usually at least two orders of magnitude shorter than in the background.

Some incidents might require a background service to be interrupted. In the drilling example which was mentioned earlier, a drill might become damaged and the assembly process might need to be stopped immediately. It is important to note that this is a different interruption than the ordinary interruption of background activities to provide communication in the foreground.

In this unexpected case, the background activity "drill three holes" needs to be stopped. It might then be replaced by "change drill #2". As soon as the drill is changed, the old background activity can be resumed.

The above example demonstrates that background activities of two priorities are desired. The regular ones, like "drilling three holes" and those of higher priority, like "stop" and possibly "change the drill". Background activities of higher priority should interrupt the current activity and should be executed immediately.

2.2. Conclusions from the Specifications

The following section corresponds directly to section 2.1. and suggest a general, hardware independent setup. As a conclusion, a fifth subsection presents a state description.

2.2.1. <u>Conclusions from the Autonomy of each Unit</u>

Each TPU should comprise sufficient resources to conduct application services. These resources have both hardware and software components. Hardware components include a CPU as computing facility, a program memory, a data memory, an interface to the application, and an internal bus. These hardware components are commercially available as controller units and typically have the size of a single board computer (SBC).

A TPU can be implemented with a commercially available SBC. The general morphology of a typical TPU is depicted in Figure 2-1. The Central Processing Unit (CPU) fetches information from the program memory, data memory, and from peripheral units via an internal bus. This information is processed in the CPU and results in data exchange with the peripheral units. The peripheral units are connected to a special application interface to provide the desired application service.

These components are available on many SBC systems. However, in addition to the application interface, another interface to the network needs to be provided. It is crucial for the autonomy of the TPU to keep these two interfaces physically and organizationally separate.

2.2.2. Conclusions to ensure Communication among Units

Establishing and conducting proper communication in the sense of a modern, versatile, and distributed control system involves two issues. A layered communication approach and the definition of a particular command syntax.

2.2.2.1. The ISO OSI Reference Model

ISO developed the OSI as a guideline for the design of communication networks. The OSI model separates the objective of ensuring communication between one or more network nodes into seven logical functions, or layers. Each layer functions independently and shields the layer above it from implementation details mastered in lower layers. As a result, the model is highly modular and flexible and encourages open networks. Each layer enjoys a great deal of independence concerning the implementation of its function, which allows application of equipment produced by various vendors [5,6,17-19]. The seven layers are defined as follows, highest level first:

Table 2-1

ISO OSI Reference Model

Name	Layer	Function
Application Layer	7	Application services
Presentation Layer	6	Code conversion and format conversion
Session Layer	5	Establishment and
Termination		
		of logical connections
between		
		nodes
Transport Layer	4	Maintenance of data
integrity		
		between nodes
Network Layer	3	Information routing
Data Link Layer	2	Transfer of data packages
Physical Layer	1	Transfer or reception of a bit stream

The IEEE 802 Standards for LANs adopted the ISO open systems concept and explicitly defines the specifications for the data link layer and the physical layer.

In the Taskmaster environment, only four of the seven OSI layers need to be implemented. There is no need for a network layer since all nodes are already physically linked by the bus. A session layer is not necessary since multi-session applications are not implemented. Lastly, a presentation layer is not necessary because neither code conversion nor data reformating need to be utilized in the environment.

The remaining four layers, the Application Layer, Transport Layer, Data Link Layer and Physical Layer have the following characteristics. The Physical Layer receives incoming data packets. The Data Link Layer passes them over to the Transport Layer, which disassembles and processes them in a form acceptable to the Application Layer. Thus, the foreground activity of the TPU concentrates on the implementation of the Data Link Layer and the Transportation Layer. It provides an intelligent interface between the issuing of commands to the network (Physical Layer) of a particular format and specified applications (Application Layer).

Compatibility of different TPUs can be achieved if each TPU understands a common syntax of foreground commands. Modularity can be accomplished if the developed unit can replace system units which realized the two layers in previous designs. This is desired in an effort to keep the Taskmaster system updated with most recently available hardware elements.

2.2.2.2. Definition of a Command Syntax

Another important issue to assure proper addressing and activation of selected TPUs is the definition of a particular command syntax, called a protocol. In earlier versions of the Taskmaster System this protocol was called Host Command Packet (HCP). Essential elements of this protocol are delimiters (to embrace the command in an effort to make it distinctive from perturbations), the network address of the target node, and the command itself. The command may initiate execution of designated pieces of code, known as tasks, in addressed TPUs. It may also contain additional information for storage in the TPU memory for use at a later time.

2.2.3. Conclusion from Command Accumulation and Automatic Recalling of Services

As specified above, recalling of services in specified TPUs requires that a special buffer is implemented on these units. All auxiliary actions associated with the service that is to be recalled are stored in this buffer. Two issues allow efficient access to these auxiliary actions. The organization of these auxiliary actions in a specially encoded task structure and the implementation of the buffer as a queue.

2.2.3.1. The Task Structure

A typical control application service can be subdivided into smaller subservices or auxiliary actions. These auxiliary actions are called tasks. An efficient way to specify tasks is a special task structure. A task is defined as an independently scheduled and programmed event which requires and services system resources [3]. In the Taskmaster System, these resources are the hardware interfaces of the TPU. A task is a modular piece of software, written in a language appropriate for the TPU and the application. For time critical actions this is usually an assembly language. Tasks are utilized to accomplish control functions by employing TPU resources.

These tasks are similar to subroutines, but can be used in a more flexible manner. Their main advantage is that in contrast to subroutines, the time and mode of task execution can be fully controlled. Tasks are represented in two formats. The first format is the command syntax described earlier. By using ASCII characters, it allows foreground commands to be issued to addressed TPUs via a communication network. These commands contain the task name and some additional information regarding the mode of task execution. The information is then compressed and stored in a buffer in a format suitable for the addressed TPU.

The main advantage tasks have over subroutines is that the task structure allows a currently running job in any TPU to be modified quickly and at any time. A great deal of flexibility can be reached in controlling applications by issuing an appropriate sequence of task commands over the network to specified TPUs. Five pieces of information are sufficient to define the execution of a task: An address, a condition, a name, a dismissal, and possible parameters.

1. **Address:** The address defines the TPU which is intended to execute the task.

2. **Condition:** The Condition defines the circumstances necessary to begin execution of the task.

3. **Name:** The name of the task identifies the particular piece of code associated with the execution of the task.

4. **Dismissal:** The dismissal specifies the destiny of the task upon its completion.

5. **Parameters:** The Parameters can be used to provide additional information concerning the mode of execution or to provide data necessary to execute the task.

Execution of a task will be initiated if two conditions are satisfied. First, a complete task specification is present in the TPU. Second, other particular conditions necessary to start the task are present. For example, some tasks may need a special synchronizing command to initiate their execution. Referring to its name, the desired task can be found in a special task library. Some tasks may use parameters, which can be part of the task specification. Upon completion of the execution, some tasks may be left in the buffer for later use.

2.2.3.2. The Queue, a Special Buffer in a TPU

The buffer as a resource of each unit can be implemented as a queue. A queue is a dynamic piece of memory realizing a first-in-first-out (fifo) organization. The queue is used to store tasks, which were issued earlier as a foreground command for later use in a background activity. They are stored in a form acceptable for the TPU. The queue permits accumulation of tasks and delayed execution. In the foreground activity, command packets representing tasks are constantly stored in the queue.

When system resources are available, the task which was received first is executed. The other tasks are executed in the same order in which they are received, provided system resources are available and conditions of execution are given. Because of the queue it is possible to specify in a command that the corresponding task, upon its completion, be placed on the queue again. Then, this task will be recalled automatically as system resources become available. Whether a task is to be requeued or not is specified in the task structure format as the "dismissal".

2.2.4. Conclusions from the Interruption of Services

The specifications discussed in subsection 2.1.4., lead to two levels of background activities. Subsection 2.2.2. describes how particular background activities can be invoked by a specific command structure. Finally, subsection 2.2.3. suggests how to classify these background activities in the task structure.

Interruption of tasks in TPUs requires a command hierarchy with at least a two-level priority. This allows interruption of the current activity of a network node to favor activities with a higher priority. The low-priority task is interrupted until the completion of the high-priority task. Then the previous task resumes.

The priority of a task needs to be issued as a part of its corresponding command. Following the task structure which was presented in subsection 2.2.3., the priority can be defined as the "Condition" (see Section 2.2.3.). Possible conditions in a two-priority structure are "Queue" and "Immediate". "Queue" means that the corresponding task is of lower priority and should be placed on the queue. It stays on the queue until system resources become available. "Immediate" means that the corresponding task is of higher priority and should be executed immediately.

An Immediate task will interrupt a background activity with priority "Queue". However, it will not interrupt a background activity of priority "Immediate".

2.2.5. <u>State Description</u>

The conclusions of the above sections can be realized with a correct software layout of the TOS. This section presents a guideline for the realization of an effective TOS utilizing a state description. A State is defined as a condition or a set of conditions of a system. In the context of the TOS these conditions refer to a particular mode of operation. In each state a set of activities is performed which distinguishes an individual mode from others. Due to the nature of the taskmaster system, a state description is appropriate. It breaks the system specification into several subspecifications. These subspecifications are more compact and easier to implement by a particular piece of software.

The system needs to keep track of any changes of these conditions in order to be able to define its state at any time. The current system conditions should be stored in a dedicated piece of memory, the system status word. Specified incidents will change these status words and thus redefine the system's state. A State Diagram [20] symbolizes these states as circles and the state transitions as arrows labeled with the respective incident that caused the transition. The State Diagram of the TOS is depicted in Figure 2-2. It comprises the following five states.

The IDLE State

The IDLE ("IDLE!") state does not perform any particular assignments. Its main purpose is to provide a defined state for power up, reset, and the lack of particular incidents which lead to other, more specialized states. The system constantly checks for incidents that require the state to change. In the IDLE state, two incidents stipulate a change. The reception of a character by the TPU or the fact that the queue is not empty. This state may be implemented as a main loop, constantly checking the status word.

The SSIR State

The SSIR ("Service the Serial InterRupt!") is defined by the execution of a serial interrupt service routine. This routine should be activated if elements of a foreground command packet are present on the network. This state comprises code to recognize, receive, interpret, and to forward command packets.

Command packet recognition involves the recognition that a character was issued to the network. Command packet reception encompasses code to determine if a particular received character obeys the given format and was addressed to the unit (myaddress?). Command packet interpretation includes the decoding of the command packet and a subsequent modification of the status word. The desired action can then be performed in specialized states. The SSIR state can be entered from any state except the TIME state. The SSIR is a temporary state. The system resumes the previous state automatically upon completion of the SSIR state.

The EQTA State

In the EQTA ("Execute a Queue TAsk!") state, the first task in the queue is fetched, interpreted, and executed. Upon completion, the task is either requeued or discarded, depending on additional information associated with the task. Provided no other special conditions are present, the system then returns to IDLE.

The EITA State

In the EITA ("Execute an Immediate TAsk!") state, high priority commands are interpreted and respective code is executed. The EITA state is always entered from the SSIR state. Upon completion of the high-priority task, the system will resume the idle state or the EQTA state depending on which state was abandoned in favor of the EITA state. The EITA state enables the system to perform high-priority tasks at any time, regardless of its current state.

The TIME State

The TIME ("update the system TIME!") state assures correct system time regardless of the current system activity. It can be realized as a high-priority interrupt service routine. The priority of the timer interrupt needs to be higher than the priority of the serial interrupt for two reasons.

First, the system time is very important for overall system performance in a distributed environment, particularly if synchronization is an issue. Second, the system will assume the state TIME only for a very short time period and then automatically return to the previous state. This is possible, because the updating of the system time does not involve lengthy computations.

Thus, the interruption will be almost not recognizable for the overall system performance. The State TIME will be assumed originating from any previous state and, upon completion will return to the previous state.

System time enhances the real time capabilities of the system and is important for the coordinated action of several TPUs. Utilizing the system time feature, tasks can be started at a specific time. Furthermore, some tasks might be implemented as endless loops and can be aborted by a time-out condition. The time-out can then be part of the task command.

2.3. Design Considerations

2.3.1. Single-Chip Unit versus Single Board Computer Unit

The advantage of distributed systems in control applications is self-evident. However, there is still some freedom in choosing the scale of the hardware units. In microprocessor engineering two choices are apparent, single chip units (SCUs) and single board computer units (SBCUs).

SCUs accommodate CPU and the operating system as well as possibly a serial communication interface on one single chip [21-23]. In microprocessor control engineering the CPU is often referred to as the microcontroller unit (MCU). The advantage is that little space is required. On the other hand, the communication interface might not be suitable for long distances. Many microcontroller chips provide a serial input/output (I/O) port but not necessarily an RS-232C or RS-485 interface. Also, the performance of the unit is confined to the capabilities of the selected chip and modifications are limited.

SBCUs usually include a variety of chips, each optimized for a particular purpose. The CPU or MCU is implemented as one of these chips. All other chips connect to the CPU through an internal bus. SBCUs require more space than their SCU counterparts, however, they are simpler to modify and to optimize.

A SCU approach would be preferred to control the fingers of an robot arm, since distances are short, the actions of each unit are simple, but space is scarce. SBCUs would be utilized in complex applications like the control of an assembly line since the units might be modified from time to time but space is not critical.

2.3.2. **Topology**

In applications of low-cost distributed control systems, bus topology is preferred because it does not require a routing algorithm. Thus, the network layer of the OSI model is not used. Physically, a bus topology can operate without a master node. Each node is passive, but listens constantly to the network, waiting to be activated. Finally, nodes can be added or deleted easily and the malfunction of one particular node does not necessarily affect overall performance.

CHAPTER 3

3. IMPLEMENTATION

This chapter utilizes the functional description of the Taskmaster as the basis for the implementation of a revised Taskmaster system, the Taskmaster II, in a new hardware environment. The Taskmaster II consists of both hardware and software.

Section 3.1. describes the selected hardware on which the TOS was implemented. Some special features that suggested the selection of the Motorola 68C11 CPU and the evaluation board (EVB) are discussed. Section 3.2. then describes the actual implementation of the TOS in the hardware environment of the 68HC11.

3.1. Description of the Hardware Environment

The following portrays the selected hardware environment in three steps. First, some of the features of the CPU are discussed briefly and illuminated in the context of the Taskmaster system. Second, the EVB is introduced and its original purpose is discussed. And last, the utilization of the evaluation board as a TPU hardware base is explained.

3.1.1. The MCU68HC11 Microcontroller Unit

Overview

The MCU68HC11 CPU is a recent 8-bit Microcontroller Unit (MCU) which offers a variety of peripheral on-chip capabilities. It is implemented in high-density complementary metal-oxide semiconductor (HCMOS) technology. This technology tries to combine the advantages of high speed, small size, low power consumption, and electrical noise tolerance.

Peripheral Functions

The chip offers, as peripheral functions, an 8-channel 8-bit analog-to-digital (A/D) converter, and a synchronous and asynchronous serial communications interface. The asynchronous serial interface was utilized as the network interface for the TPU. The serial synchronous interface and the A/D converter are still available for application services. The MCU includes several I/O ports, one of which has parallel capabilities.

Special Features

The MCU contains a "watchdog system" that can be utilized to monitor proper execution of user developed software. A WAIT and STOP mode is available which can send the MCU to sleep in order to reduce power consumption when needed. This might be useful in battery driven applications.

Registers

The MCU includes seven primary and 64 secondary registers. The primary registers are two accumulators called A and B, two index registers, called X and Y, a stack pointer, a program counter, and a condition code register. With the exception of the accumulators and the condition code register, which are 8-bit, all primary registers are 16-bit. The accumulators can be combined to form a 16-bit accumulator D or can be addressed separately.

The 8-bit secondary registers are dedicated to a variety of different functions and are all bit addressable. Some of the registers define the operation mode of the MCU and are time-protected. They can only be modified during the first 64 instruction cycles. This prevents an unintentional change of the system setup during program execution. Other secondary registers control output pins or buffer incoming data.

Two useful groups of secondary registers are the Timer Output Compare (TOC) and the Timer Input Capture (TIC) registers. Both operate together with the 16-bit on-chip counter. TOC registers can be utilized to program an action to occur at a specific time (when the 16-bit counter reaches a specific value). TIC registers can be employed to record the exact time of external events. In the Taskmaster II application, TOC4 was utilized to provide the TPU system time. System time is defined as a clock which counts hours, minutes, and seconds, and is implemented on each TPU. The clock starts counting upon reset.

On-Chip Memories

A whole family of MCUs is available, varying mainly in type and size of available on-chip memory. Depending on the specific part type, the MCU comprises at least two of the following memory types: EPROM, ROM, EEPROM, and RAM. In this work an XC68HC11A1 MCU was used. The size of the memory varies with the part type. In the Taskmaster application, the 512 byte on-chip EEPROM might be useful to store the TPU address and possibly some discriminator information to adjust TOS application to special customer needs. The largest available ROM has 12K. This is sufficient to hold the TOS and a large library of application tasks.

Operation modes

As depicted in Table 3-1., the MCU can operate in four modes which are logically selected by the MODB and the MODA pins of the MCU. However, only the Normal single Chip mode (NSC) and Normal Extended (NE) mode are relevant in the Taskmaster application. The other modes are mainly designed for factory test.

Table 3-1

Hardware Mode Select Summary

Inputs		Mode Description	Abbreviation
MODB	MODA		
1	0	Normal Single-Chip	NSC
0	1	Normal Expanded	NE
0	0	Special Bootstrap	
0	1	Special Test	

In the NSC mode, the MCU operates as a single chip controller. All ports are available for I/O functions since no external address/data bus is needed. All software essential to control the MCU is held in on-chip memory. The NE mode utilizes two ports (port B and port C) for address and data bus functions. Program instructions are fetched by these interfaces through an external bus from external memory (see Figure 3-2).

Interrupts

The MCU includes 18 separate interrupt sources [23]. These sources include two external interrupts and five TOC interrupts. TOC interrupts can be activated if the free running counter matches one of the five TOC register values. A priority hierarchy of the maskable interrupts can be defined during system setup by user code.

Instruction Set

The 6811 instruction set is compatible with the instruction set of the 6800 but offers some additional instructions. The TOS requires frequent transfer of data sets. This can be accomplished efficiently by indexed addressing, utilizing the two index registers. In addition, a wide variety of branch instructions are available. Especially useful are the two conditional branch instructions BRCLR and BRSET. In BRCLR, a specified memory byte is ANDed with a bit mask. A branch is performed if the result is zero. Similarly, in BRSET the inverse of the memory byte is ANded with the mask. This allows program flow to be directed by the bit value of specified bits in memory locations.

Although only an 8-bit internal bus is available, 16-bit instructions are feasible by accessing two successive memory bytes subsequently. A 16-bit accumulator is available for 16-bit instructions. Arithmetic and logical shift instructions are available for 8-bit and 16-bit. Compare instruction are also usable both in 8-bit and 16-bit. Finally, integer and fractional 16 by 16 division is available and will certainly be useful in a variety of user defined application services.

3.1.2. The Evaluation Board

The Evaluation Board (EVB) is a low cost tool for debugging and evaluating of MCU68HC11 MCU-based microcontroller systems. The EVB is depicted in Figure 3-1 and its schematic is depicted in Figure 3-3. The EVB emulates NSC operation of the MCU utilizing its NE mode. This concept can be illustrated best in an example.

A typical application for operation in the NSC mode is the control of a complex application environment. The MCU then is plugged into a specific socket of the application environment's control circuitry. Software which is necessary to service the application is stored in MCU on-chip memory.

For software development, the use of the EVB is similar to the utilization of an emulation system. The EVB then connects to the socket instead of the microprocessor (or an emulation system). The EVB (similar to an emulation system) simulates all I/O operations of the microcontroller. The application environment cannot detect whether it is controlled by a microcontroller or the EVB (or an emulation system).

Software development on the EVB yields the advantage that software is stored in external RAM of the EVB rather than burned into on-chip ROM. This simplifies the development, because software in a RAM can easily be modified at any time. The layout of the EVB warrants that if the code functions on the EVB, it will also function in the NSC mode stored in on-chip MCU memory. The applications engineer can modify code which is stored on the EVB till satisfactory performance is achieved. Then the software can be installed on-chip.

One RS232C interface is implemented on the chip to connect to a PC as a development system. Development software is provided by Motorola and discussed in Section 3.2.2. On the EVB, the MCU interfaces to an external address/data bus to external memory through ports B and C of the MCU. The MCU on the EVB operates in NE mode, but the whole EVB emulates a single-chip mode operation of a MCU68HC11 MCU. A MCU 68HC24 emulation chip is utilized on the EVB to emulate the single chip operation of ports B and C, since those ports are lost due to the NE mode operation of the board.

3.1.3. <u>The Evaluation Board as Taskprocessor Unit</u>

The EVB was specifically designed as a NSC emulator for the MCU68HC11. However, comparing the EVB description in Section 3.1.2. with the specifications of a TPU, which were presented in Section 2.1., it becomes evident that the EVB includes all hardware elements of a TPU. From a hardware point of view, the EVB is a SBC with additional features.

Commercially available SBCs are tested and their functioning is usually reliable. In addition, they are produced in large enough quantities to offer a good price-quality ratio. This makes it often more efficient to modify such a board rather than to develop a new design from scratch.

The EVB offers one possibility for the implementation of an MCU68HC11 based SBC. The three interfaces of an EVB based TPU depicted are depicted in Figure 3.-2. The EVB Port P1 provides the interface to the MCU ports. It is the main interface to application services. Port P3 is an asynchronous RS232C interface to the MCU which provides communication to the network. A NIU can be utilized to connect the RS 232c Interface to a Bus Standard (e.g. RS 485). This has been shown already in other work [17]. Finally, Port 1 is still preserved as an interface to a development system as described in Section 3.1.2.

It was the intention of this work to utilize the EVB both as TPU and as low cost development tool. Both capabilities are selectable separately by the switching of jumper J4 on the EVB board. The EVB thus operates in two modes. First, in a work mode, it is a fully qualified TPU. In this setup the evaluation capabilities are of no interest and relinquished. The EVB will typically operate in this mode when all system code is developed and optimized with respect to the needs of the application services.

Second, in a development mode, each TPU is still a fully qualified EVB and can be employed to develop user code for the control of application services. In addition to the possibility of network independent EVB operation, this EVB can also stay connected to the network during the evaluation. This yields the advantage that development and optimization in TPUs can be pursued with respect to incoming network commands.

This can be very useful in the previously introduced assembly line example. The overall system might have been designed, but during tests it turns out that one TPU's control performance is too slow compared to incoming network commands. It might not be efficient to change the whole system setup. Instead, utilizing the EVB capabilities with network connection, the code efficiency on the particular TPU can be increased until hand-in-hand operation with the network is reached.

In a proposed factory environment, a system engineer would take his laptop computer to the location of the respective TPU, plug it in, and choose development mode. The laptop then functions as the development computer and is connected to Port P2 of the EVB. Incoming commands are provided by the network, or are simulated by the laptop and submitted at Port P3. As soon as acceptable performance is reached, the new code can be downloaded into the TPU. The development mode is then switched to working mode. Later, the code can be burned into a PROM and plugged into the TPU to replace the old PROM. In order to allow the above operation, Port P2 must be grounded at line seven. This is not done on the factory supplied board.

Using an EVB as a TPU yields another advantage. The EVB was designed to develop code intended to run in NSC mode. In this work, the EVB was then used as a TPU unit. Each TPU thus has the size of a SBC. The evaluation capabilities of the EVB were made available for distributed system development as described above. If it is desired to setup a distributed system with units of single-chip size rather than SBC size, this system can be achieved using the network and several EVBs. When the final TOS version is developed, it can be installed on each MCU to be run in NSC mode. Thus this system could be utilized to develop a Taskmaster system based on units of single-chip size.

3.2. Software Description

The software implementation of the revised taskmaster system, the Taskmaster II, involves two separate pieces of software. These are the TOS and the development software. The TOS emerged entirely from this work. The development software is a public domain software issued by Motorola which was intended to serve as a low-cost development tool for microcontroller applications. It was not intended to support the development of distributed systems.

In addition to the Taskmaster implementation, this work preserved the Motorola development aids and allowed their use in the more complex environment of a distributed system. Thus, these tools are still available to further enhance the current version of the Taskmaster II.

3.2.1. Taskmaster Operating System

This subsection discusses the implementation of the TOS on the Motorola 68C11 hardware environment which was described previously. Even though this TOS was specifically designed for the 68C11, some of the following considerations might still be appropriate for the implementation of a similar TOS on a different hardware environment.

The following description should be accepted in the context of the preceding functional analysis which was presented in Chapter 2. It is grouped into two subsections, the realization of special software elements and the realization of the five described states.

3.2.1.1. Realization of Special Software Elements

Five software elements require further explanation. These elements include the classification of tasks, the calling of tasks, the implemented formats, bit controlled program flow, and the realization of the queue.

Classification of Tasks

Figure 3-3 groups the spectrum of tasks distinguished by this system together with corresponding priority levels as a linked tree of tasks. Two main priority levels were implemented by discriminating Queue tasks and Immediate tasks. Queue tasks themselves can be further classified as Ready tasks and Synchronized tasks. Ready tasks will be executed as soon as they reach the head of the queue and the system is ready to execute the queue. Synchronize tasks must be invoked by a specific Immediate task and are left for later implementations.

Among the Queue tasks, the system also differentiates between Queue-Requeue and Queue-Non-Requeue tasks as tasks to be rerun and tasks to be discarded subsequent to their completion. An additional class of repeat tasks which are rerun a specified number of times could be implemented later.

Another class of tasks are Short tasks. These tasks are a special class of Immediate tasks. Short tasks are addressed to all TPUs and do not follow the common HCP format. This implies that they are executed faster than regular Immediate tasks. As a typical example for such a task, an abort task was implemented to abort all current activities in all units.

Calling of Tasks

Calling of tasks is realized in the states EQTA and EITA separately according to their respective priorities. This permits calling of a high-priority task even during the execution of a low-priority task. The general approach in calling a task, however, is the same. All tasks have a hexadecimal number in the range of 00H to FFH as a name. Thus, reference to a particular task can be accomplished by a brief calculation in a specialized vector routine. Upon completion of the task, program flow continues at the location succeeding the call of the vector routine, regardless of the executed task. All information necessary to vector a task is gathered from the current system status, represented by three status words (see "bit controlled program flow" in this section).

Task Formats

Two formats are present in the Taskmaster. These are the Host Command Packet (HCP) format and the Assembled Command Packet (ACP) format. The HCP is the format of the foreground commands which are issued by the Host computer or possibly other units. The HCP format is present on the physical layer, in other words, the network. The HCP format corresponds to the task structure and its elements which were described in Section 2.2.3.1. It embodies the following ASCII characters, further specified in Table 3-2.

{AA P NN S DD DD DD DD DD}

Table 3-2

Host Command Packet Format

Element	Meaning	Example
{	Begin delimiter	
AA	TPU hexaddress	01
P	Prefix	:
NN	Tasknumber	C3
S	Suffix	.
DD	Data	3E
}	End delimiter	

The begin and end delimiters are necessary to envelop the contents of the foreground command in order to distinguish it from noise or other activities on the network which are not related to a command.

The Prefix describes the conditions necessary to execute a task. This includes the priority level of the tasks. The legal prefixes are listed in Table 3.-3.

Table 3-3

Legal Prefixes

Prefix	Meaning	Priority Level
:	Ready Queue task	low
?	Synchronized Queue task	low
!	Immediate task	high

The address, in hexadecimal, distinguishes the particular TPU which was selected to run the task.

The Suffix corresponds to the dismissal which was specified in Chapter 2. It defines the destiny of the task after its execution. Table 3-4 lists the legal suffixes.

Table 3-4

Legal Suffixes

Suffix	Meaning
.	Discard upon completion
+	Requeue upon completion
*	Requeue a specified count of times

Discarding and requeuing is implemented in this work, requeuing a specified number of times is only suggested.

Arguments can be sent as part of the command. This is desired for some tasks. For example, to set the time, variables representing hours, minutes, and seconds can be included in the command as arguments. The current setup allows 5 arguments. This can easily be extended to 7, 15 or any number compatible with the device's RAM storage.

Each TPU only accepts a proper HCP format addressed to itself. The TPU/TOS converts the HCP format into an Assembled Command Packet (ACP) format. The conversion from HCP to ACP format is done because the HCP packet consists of ASCII character elements. These are not appropriate for efficient storage or processing.

Since the address is correct, it is no longer needed and discarded. The ASCII task number is converted into a hexadecimal format in order to save memory. The two ASCII characters representing suffix and prefix are bundled in a bit pattern as a Task Status Word (TSW). This TSW is implemented as one byte. In addition to suffix and prefix information, it contains the number of arguments which were previously submitted in the HCP format. In other words, the TSW contains sufficient information to entirely define the operation mode of a task.

The arguments of the HCP format are also converted from ASCII into hexadecimal format. In total, a five argument task command requires 18 bytes in HCP format and seven bytes in ACP format.

Bit Controlled Program Flow

The TOS activity can be grouped into five states. Each state is implemented as a distinctive piece of code and described in section 3.2.1.2. Operation in each state varies according to the system status. The system status can be viewed as a record that specifies what has been accomplished by the TOS at any given time. These accomplishments determine operation details in each state. In addition, they might cause entry into a different state.

The system status is defined by two system status words, SSW1 and SSW2, each of which comprises 8 bits. The relation between the bits of the status words and the five states of the TOS is described in section 3.2.1.2. The Tables 3-5, and 3-6 present a bit description of SSW1 and SSW2. Column "Meaning" always refers to "bit set". Some bits are still available for future implementation. They are labeled as "f.f.e.", meaning "for future extension". The tables also show the reset value of each bit in column "Res".

The system status will be modified depending on the nature of the currently processed task. These specifications are stored as TSW. The TSW is part of the ACP format and always attached to a task for internal processing management. Table 3-7 presents a bit description of the TSW.

Table 3-5

Bit Description of SSW1

Bit	Name	Res	Meaning
7	requeue	0	Requeue task upon completion
6	no-interrupt	0	execution of running task cannot be interrupted
5	time-out	0	abort due to time-out allowed
4	synchronize	0	task requires synchronization
3	pause	0	current Queue task is paused
2	Queue task	0	processing of a Queue task
1	Immed task	0	processing of an immediate task
0	queue empty	1	task queue is empty

Table 3-6

Bit Description of SSW2

Bit	Name	Res	Meaning
7	error	0	can provide error
6	f.f.e.	-	for future extension
5	f.f.e.	-	for future extension
4	f.f.e.	-	for future extension
3	address check	0	HCP address check in progress
2	HCP overflow	0	HCP overflow occurred
1	echo	0	echo previously received HCP
0	HCP progress	0	a HCP assembly is in progress

Table 3-7

Bit Description of the TSW

Bit	Name	Res	Meaning
7	synchronize	0	task requires synchronization
6	Requeue task	0	requeue task upon completion
5	Repeat task	0	requeue task as many times as specified in first argument
4	no-interrupt	0	execution of running task cannot be interrupted
3	f.f.e.	0	for future extension
2	arg c.2	0	argument count in binary digits, position 2
1	arg.c.1	0	argument count in binary digits, position 1
0	arg.c 0	0	argument count in binary digits, position 0

Realization of the Task Queue

The task queue is realized as a memory page whose top and bottom edges are cyclically linked by software. Virtually, neither beginning nor end exists. Memory entry of data is administered by an entry pointer and a length count.

The length count keeps track of the amount of stored memory. Data is stored in the location which is "pointed to" by the entry pointer, provided that memory space is available. The entry pointer is automatically advanced with each entry. If it exceeds the lower margin of the physical memory it is wrapped to the top of the assigned memory space.

This setup allows a first-in-first-out (fifo) service of tasks which are stored on the queue as data entities. Tasks that came in first, will be serviced first as soon as system resources become available. Entry pointer and length count provide enough information to determine the head and tail of the queue as well as the availability of memory at any time.

3.2.1.2. <u>Software Description of the Five Basic States</u>

This section will present the realization of the five basic states as separate software modules. The system status words and the task status word (see 3.1. and Tables 3-5, 3-6, and 3-7) define the execution details in each module and also initiate switching of the program flow among modules. This presentation is illustrated by flowcharts. Since they are close to the state description which was presented earlier, Flowcharts were selected for program documentation.

The IDLE State

The IDLE state is entered following a reset and system initialization. As depicted in Figure 3-4, the IDLE state comprises an endless loop. For test purposes, the least significant bit (LSB) of Port B outputs a square wave with a period of 10 ms which can be observed by an oscilloscope. The endless loop is executed as long as no serial interrupt is furnished and the task queue is empty.

The endless loop also includes a window for a serial interrupt. This serial interrupt is activated by the reception of an ASCII character through the network. Program flow then switches to the SSIR state temporarily to service the interrupt (see "the SSIR state" in section 3.2.1.2.). If no serial interrupt occurs, but the task queue has been filled, program flow switches to the EQTA state (see "the EQTA state in section 3.2.1.2.).

The SSIR State

Figure 3-5 shows the rough features of the SSIR state which provides the serial interrupt service. It is important to note that state SSIR can only service one character at a time because characters of the HCP format are received sequentially and asynchronously. At the very beginning additional serial interrupts are inhibited. The TOS now fetches the received character from a buffer for further examination.

If the received character is a "begin" delimiter, HCP processing commences. If HCP processing has already begun, the received character may represent a Short task or special service. In that case, the respective Short task or special service is executed immediately. Else, an address check is performed if necessary. As soon as the address has checked out successfully, subsequently received characters are considered as elements of the HCP packet and are stored in a buffer if they are legal. If the address check was not successful, processing of the HCP is discontinued.

As soon as the character of the task (Queue or Immediate) becomes evident, further processing continues separately. In both cases the HCP format is converted into an ACP format. However, in the case of an immediate task, program flow is directed to state EITA which executes the task immediately. In the case of a Queue task, the task is stored on the queue and program flow exits state SSIR.

Finally, at all exits of SSIR, the serial interrupt is enabled. The serial interrupt is restrained temporarily in order to prevent stack overflow due to subsequent serial interrupt service requests. Since state SSIR, in comparison to other states, is assumed only for a very short time period and since the 68C11 asynchronous serial input port is double buffered, it is unlikely to loose a character.

The EQTA State

Figure 3-6 presents the implementation of the EQTA state. The first task in the queue is fetched and its ACP format is interpreted. If this is a Synchronized task, program flow is paused until a synchronize command is furnished through a serial interrupt. When the task is ready, program flow is directed to a particular piece of code that represents the task. The task is then executed.

Upon completion of the task, the task is either requeued or discarded, depending on the corresponding task status word. Program flow then continues in state IDLE.

The EITA State

The EITA state operates similar to the EQTA state, but it executes an Immediate task instead of a Queue task. The corresponding piece of software is depicted in Figure 3-7. Since it is an immediate task, it is supposed to be executed instantaneously. No further considerations as for synchronization are necessary.

An Immediate task cannot be requeued or interrupted by any other task. It is executed in the serial interrupt service routine which is implemented by the SSIR state. The EITA state can be considered as a substate of SSIR state. Upon completion of an Immediate task, program flow continues in the SSIR state.

The TIME State

System time is implemented by three dedicated memory bytes which represent hours, minutes, and seconds. Every 25 ms an interrupt is generated which increments a counter. This counter is utilized to update the "Seconds" byte precisely every second. Similarly, "Seconds" are updating "Minutes" and "Minutes" are updating "Hours" as shown in Figure 3-8.

A special task can be written to set the system time. In this implementation, the system time starts at zero when a reset is furnished. It is important to note that the TIME state is an interrupt service routine with the highest priority. It is called automatically by the TOS system every 25 ms, no matter in which state the TOS is currently operating. Thus proper system time is always secured.

If all TPUs are reset in a coordinated fashion, then they all will have the same time reference. This can be accomplished by issuing a short task "reset" over the network. A common time reference for all TPUs is important for coordinated action.

3.2.2. Development Software Elements

In this subsection, software to utilize the EVB as a user code development system is described. This includes possible modification of the TOS or addition of some user designed tasks. Although employed for development purposes, the corresponding software was not developed in this work and shall therefore be discussed briefly. Respective manuals offer further information [22,23].

A Communication Program

A communication program is utilized to establish and maintain communication between the development system and the TPU as target system through an RS232C interface. This communication program is run on the development system which is usually implemented as a PC. Two commercially available programs can be used as communication programs, PROCOMM and KERMIT. Detailed description is available by the respective software distributors and in the user's manual of the EVB [22].

Utilizing the communication program, commands can be issued from the PC to the TPU in order to control the monitor program. Likewise, code which was developed on the PC can be downloaded into TPU RAM.

The AS11 Cross-Assembler Program

The AS11 cross-assembler converts assembly code into executable code in Motorola S19-format. It is provided by Motorola together with the EVB for the use on IBM-compatible PC/ATs. Using the MCU68HC11 instruction set [23] and a texteditor, the user can write an assembly program and have it compiled by the cross-assembler on the PC. The cross-assembler provides a listing and possibly error messages if compilation could not be accomplished successfully.

The Buffalo Monitor Program

The Buffalo Monitor Program is intended as low-cost tool for debugging and evaluation of MCU68HC11 MCU-based target system equipment [22]. The monitor is an assembly program, intended to be installed in a PROM on the EVB. Provided that proper communication is established between both systems, operation of the monitor can be directed through commands issued by the development system. The monitor operates in two different modes, a debugging mode and an evaluation mode.

Debugging Mode of the Buffalo Monitor Program

In the debugging mode, the monitor can download executable code which was assembled on the development system and help debugging it. Some commands are available to show or modify register or memory locations, and to perform single-step execution of the downloaded code. In this mode, the monitor can also download executable code or assemble code to executable code line by line.

Evaluation Mode of the Buffalo Monitor Program

In the evaluation mode, downloaded code can be executed. This allows the user to evaluate the performance of developed code in a target system environment because the EVB emulates NSC operation of the MCU. Upon completion of some initialization procedures, the monitor program relinquishes control in favor to the user program. User code that performs satisfactory in the evaluation mode will perform in the same manner when implemented on the MCU provided proper MCU system initialization is included in the NSC version of the code. This can be accomplished in a short setup routine.

One property of the monitor is that it provides system setup of the MCU. This has the advantage that code run in an evaluation mode rather than without the monitor at all is relieved from some system initialization procedures. However, using the monitor does not allow influence of a variety of system parameters since their setup might be time-protected. Unfortunately, during monitor execution, one output compare register (TOC5) and a significant part of the on-chip RAM is not available for user application [22,23]. Thus, these resources cannot be used in the developing user program. However, still four other output compare registers and 54 bytes of on-chip RAM are left for the user's disposal.

CHAPTER 4

4. EVALUATION OF THE TASKMASTER II

This Chapter critically evaluates the accomplishments of this work. Section 4.1. summarizes the system's capabilities as an interface between human creativity and bit level activities. Section 4.2. discusses to what extend previously stated specifications were met and how the performance of the TOS core was tested with a selection of implemented tasks; it also mentions advantages of the system over previous implementations. Section 4.3. covers drawbacks.

4.1. Conclusion

General Discussion

From the user's point of view, Taskmaster II efficiently interfaces human control desires to bit level activities of distributed system resources. Physically, control is performed as close to the application environment as possible. Specialized units, the TPUs, are implemented at the location of the application environment and perform control in a background mode. In addition, each TPU is able to communicate over a communication network in a foreground mode.

The utilization of the task structure allows the system to be controlled by issuing high level commands and to schedule predefined activities, known as tasks. The skeleton of each task is accessible in each TPU. Operation details of tasks are issued in the high level command. Scheduling of control activities can be performed by a human operator or by a computing facility, the SS. The SS is capable of administering a task schedule, utilizing the task structure and the task command format (HCP format).

Figure 4-1 illustrates the cooperation of elements in a control environment utilizing a Taskmaster system. Each element, the application, the TPUs, the SS, and the human operator is symbolized as a shell with assigned responsibilities. It is important to note that the abstraction level performed in each layer increases from the center (the bit level activities), up to the outmost shell (human creativity).

In addition, the illustration shows how low level activities performed by the application environment are shielded from the human operator by the elements of the Taskmaster system, SS and TPU. The human operator formulates the control problem specification as a control algorithm. This algorithm is implemented as a high level language program and runs in the SS. The high level program communicates to the TPUs via foreground commands, utilizing the HCP format. The TPUs themselves control the application via assembly code and background activities, some of which include bit level activities. Furthermore, the outmost shells, human operator and even SS can be removed, and the system is still able to perform control assignments autonomously.

Discussion of the Character Processing Time

Background activities are interrupted in favor to process characters which have been received in the foreground mode. In order to evaluate system performance, the ratio of the time needed to process a character and the time remaining to service background activities is important. In this reasoning, a worst case is assumed in which a constant stream of characters is sent.

The Baud rate determines the time period between two subsequently received characters. The TOS needs to operate fast enough so that the last received character can be processed before the next character is received. If upon completion of the character processing, time is left until the arrival of the next character, this time can be utilized for background activities. The length of the processing time of a character depends on the software design and the given oscillator frequency.

In this work, operation with a transmission rate of 9600 Baud was successfully tested. State SSIR represents the service of a received character. The necessary processing time varies with the received character. An end delimiter "}" requires the longest processing time if it represents the end of a Queue task command. State SSIR then needs to assemble the ACP and to place it on the queue. In the current setup this requires approximately 600 CPU cycles which equals 0.3 ms since the CPU is driven by a 8 Mhz clock.

Since each ASCII character is sent as a stream of 10 Bits, with 9600 Baud, a character arrives every milisecond. At 9600 Baud, even if every received character was an end delimiter, 1/3 of the available time is needed to process the character. The remaining 2/3 are available for background activities.

Even more, this assumption is much too pessimistic. Since a HCP representing a Queue task comprises at least 13 ASCII characters (including the delimiters), an end delimiter "}" can occur only every other 13 characters. The processing of other characters varies between 0.015 ms (erroneous character) and 0.15 ms (average legal character). The processing time for legal characters varies between 0.11 and 0.2 ms. This is a little faster than the old system where the average character processing time was about 0.21 ms. Further improvements will most likely require a second processor which is entirely devoted to communication activities in the foreground.

Under the assumption that a constant stream of commands is sent without pause and with a transmission rate of 9600 Baud, it can be concluded that approximately 70 percent of TPU computing time is available for background activities.

4.2. Advantages of this System over Previous Systems

The advantages of this system can be grouped in three subsections. These are advantages due to the selected MCU, advantages due to the implementation of the TPU as EVB, and advantages due to the developed TOS.

4.2.1. Advantages due to the Selected MCU

The advantages of the chip resulting from the properties of the MCU were described in section 3.1.1. Many of those features were not needed in the Taskmaster implementation. However, some may become very useful for the utilization in TPU application tasks. Among these features, the analog-to-digital converter system, the parallel I/O port, and the synchronous and asynchronous I/O ports are attractive for advanced control applications.

4.2.2. Advantages due to the TPU Implementation as EVB

Chapters 2 and 3 deducted that TPUs can generally be implemented as SBCs. The EVB is a SBC, specialized for development and evaluation purposes. In other words, it is a low-cost emulator. In spite of the specialization, it still has the basic morphology of a SBC and thus can be used as a TPU. Utilizing the EVB yields the advantage that, on top of all TPU related functions, the emulation feature could be preserved in this work.

The EVB was utilized in three modes. These modes are regular TPU mode, regular EVB mode, and a combined mode. The main achievement of this work was the implementation of a versatile Taskmaster system, utilizing the EVB as a TPU in the first mode. However, it still allows the utilization of the EVB as emulator in a second mode. In addition, in a third mode, the EVB allows the development and evaluation of the TPU operation itself. This transferred the evaluation idea from one SBC to a distributed environment. This last mode allows the decision to optimize TOS code located in particular TPUs with respect to the reception of foreground commands from the communication network.

As a result, this work presents a fully functional, revised Taskmaster system with additional software development capabilities. Implemented in distributed control applications, this allows a quick and flexible response to upcoming modification needs by adding, deleting, or modifying tasks in the task library. These modifications can be accomplished quickly and efficiently since sufficient development aids are readily available on each TPU.

A second advantage from the implementation of the TPU as EVB is the emulation of SC operation of the MC68HC11 MCU. This supports the development of a SCU based Taskmaster system. Code in each TPU thus can be optimized until the whole system functions satisfactory. At this point, the realization of a SC based Taskmaster system is simple since downscaling of each TPU to single-chip size is straight forward by utilizing the EVB's emulation capabilities.

4.2.3. Advantages due to an Improved TOS Core

Advantages corresponding to this subsection are grouped in Structuring, Documentation, and Performance

Structuring

Due to the preceding functional analysis, a highly structured top-down approach could be pursued successfully. A state description was derived and a hierarchy of tasks was suggested. Both provided a clear line of authority. The entire system administration is concentrated on three 8-bit status words. These status words clearly define operational details in each state and serve as a basis for state transitions.

The previous Taskmaster design was reconsidered as a result of the above reasoning and adopted to the new hardware environment. The two index registers of the 68HC11 MCU allowed a pointer-based TOS design leading to some simplifications in memory transfer and to enhanced program readability.

Documentation

This work presents a functional analysis of the Taskmaster and a general, hardware independent guideline for the implementation of the most recent TOS version, the Taskmaster II. Relevant Taskmaster features are described in detail. In addition, the implemented TOS program is documented by comments in order to enhance readability. This supports improvements in two different levels.

In a first level, the Taskmaster system can be improved by modifying or by adding tasks in the task library. This requires some knowledge of assembly programing but no deeper insight into the Taskmaster system. Adding and optimization of particular tasks can be appropriate projects of further work. In a second level, the TOS core can be modified. This requires an in-depth understanding of the TOS details.

Performance

This work concentrated on the implementation of the TOS core as a distributed operating system. The task structure was implemented to administer dedicated pieces of code, known as tasks. Following the guidelines of the functional analysis, all relevant features like a queue, system time, and two priority levels of tasks were implemented. The overall performance was tested with an appropriate variety of tasks as follows.

Functioning of the Queue

Four tasks, each of which sends one single ASCII character have been included in the task library of the TOS in addition to an Endless task to produce a square wave at an output pin. It could be shown that these tasks can be placed on the queue with a "Reque" or "Discard" option. When the ASCII tasks were loaded on the queue, the system kept on sending the corresponding characters. When the Endless task was called, queue processing stopped since this task never ended.

Cooperation of Foreground and Background Activities

While processing the queue the system is either in the EQTA state or the IDLE state. Thus, it is either currently executing a Queued task or has just been executing a Queued task. It could be shown that even while currently executing a task in the background activity, commands were accepted in a foreground activity at any time. As an example, while executing the Endless task, new commands were still accepted and corresponding tasks could be queued.

Functioning of the Higher Immediate Task Priority

At any time, even when currently executing a Queue task, Immediate tasks were accepted and executed immediately, while an eventually running Queue task was interrupted. Upon completion of the Immediate task, execution of the Queue task resumed. As examples of Immediate tasks, the contents of the queue and the current system time were displayed while the ASCII tasks were processed, or while the Endless task was running.

Functioning of the Short Task Feature

As an example of short tasks as Super Immediate tasks, an Abort task was successfully tested. When the Endless task was currently running and using system resources, the Abort task could abort the Endless task and allow queue processing to resume. If other tasks were loaded in the queue, these were then called sequentially.

If the Endless task was queued as Queue-Reque task, then of course, the system locked up again after having processed all other tasks in the queue. However, the Endless task could then be aborted again. The Abort task would also be functional as an addressed Immediate task.

The above showed proper functioning of the entire TOS system. All suggested tasks of the developed hierarchy were implemented in at least one example. It could be shown that the system is functioning in all states as intended.

4.3. Drawbacks of the System

Two major drawbacks of the system need to be stated. The set of tasks is not complete and the system is currently not set up for running without the monitor chip which is provided on the EVB by Motorola. The following discusses these drawbacks and gives suggestions to fix them.

Incomplete Set of Tasks

The TOS currently lacks three Short task features, which were implemented on previous systems. These are the "Pause" and the "Resume" tasks, and the synchronization feature. The "Pause" task is similar to the "Abort" task. However, instead of a final abort, the task is interrupted and can be resumed with a "Resume" task. These tasks can be easily written in a "copy-and-modify" approach using the already provided "Abort" task.

Other important basic tasks that are not yet implemented include a task to delete specific tasks on the queue according to their name or their entry number. This would further enhance the versatility of the Taskmaster system. A task to set the system time is also currently lacking and might be useful in some applications. Finally, it might be desired to have endless tasks run a prespecified amount of time and be terminated by a "Timeout". This, as well is not implemented in the current version.

A task to delete particular tasks on the queue is relatively simple to write. Sufficient administration is already provided to keep track of the "fingerprints" of all queued tasks. At any given time, it is possible to determine the name and status word information of each queued tasks. Additional information is present about the character of currently running tasks. A task to display the queue is already implemented. This gives enough data at any time for a particular Immediate task to delete a selected Queued task referring to either its name or its entry number. With similar background information this has already been shown in other work.

Running a task a prespecified amount of times requires the TOS to keep track how many times it has already been requeuing a particular task. This can be realized by setting a dedicated counter at each requeuing event. When a particular count is reached, further requeuing is inhibited. Setting of the system time is simple since it only requires transfer of arguments from the command to particular memory locations.

To have a task run to a prespecified "Timeout" is only slightly more sophisticated. It requires the time-out time to be specified by another task. Then, in addition to performing the actual task, this task needs to compute the absolute time-out time and constantly compare it to the system time. This can most easily be realized by using one of the remaining Timer Output Compare (TOC) registers. Such a register can be set up to issue a particular action when the register contents matches the contents of the free running 16-bit counter. Auxiliary registers can be employed to cover time periods exceeding the time which corresponds to one 16-bit counter revolution. This technique can be copied from the already implemented approach to generate a system time, which utilized the TOC register #4.

TPU operation currently requires plugged-in monitor PROM

The evaluation features of the EVB are handy during development of a system. It is fully justified to tolerate a particular PROM which is entirely dedicated to the emulation and debugging on the TPU. However, when a final version of the software is completed, this debugging feature is no longer necessary. It may then be removed completely from the board. This, however, is not possible in the current setup.

It is possible to switch with jumper J4 between a Development Mode and a Free Running Mode. In the Free Running Mode, only a small portion of the monitor is executed. Subsequent to some system setup, the monitor detects the hardware condition determined by jumper J4 and then relinquishes control in favor to the application program, in this case the Taskmaster. Due to the current hardware and software setup, it is necessary that this small portion of the monitor program is executed.

However, with jumper J4 set up as "Free Running", the Taskmaster can be started upon reset because it is burned in a separate PROM. No interaction with the monitor is necessary. The Taskmaster can be invoked upon reset. As an actual drawback, memory space, which could be used for applications, is unnecessarily blocked by an unneeded monitor.

If desired, this can be fixed by simply extracting the setup routines from the monitor and by adding them to the Taskmaster code. In addition some parts of the setup might be modified. On the other hand, memory space is not scarce on the board since, in addition to the Taskmaster PROM and the monitor PROM, there is still 8k of RAM left. Only a fraction of this RAM space is used by the taskmaster for the queue and some other temporary storage. The major part is available for application use.

Advantages and drawbacks considered altogether, it can be concluded that the new Taskmaster System represents a successful implementation and updating of the Taskmaster idea. Previous achievements were selected and included in the context of a fresh approach and a new hardware environment. A sound and versatile system emerged and was tested successfully. Room is provided and incentive is given for further enhancement of the Taskmaster II.

CHAPTER 5

5. SUGGESTIONS FOR FURTHER RESEARCH

This work is the first implementation of a Taskmaster system based on a Motorola MCU. The main TOS was successfully updated based on the MCU68HC11. However, as presented in Section 4.3., some enhancements of the system are still possible. This Chapter submits some suggestions for possible improvements in a variety of areas.

Completion of Auxiliary Functions

Auxiliary functions which were already developed in other Taskmaster systems might be included to further broaden the versatility of the system. Examples are a "pause" and a "resume" feature, to interrupt and restart currently executed tasks. In addition, a "repeat" feature as a variation of the "requeue" feature can be added. "Repeating" would be defined as "Requeuing" a specified amount of time.

Additional Priority Levels

In addition to the two main priority levels "Immediate" and "Queue", the priority of some Queue tasks could be elevated by inhibiting their interruption by Immediate tasks.

Enhancement of the Task Library

Many tasks for system administration or applications can be added to the task library. Tasks to modify the TSWs of tasks on the queue or to rearrange the order of their execution are most valuable. Other important administration tasks can be devoted to provide assistance in delaying of tasks or in coordinating their execution in specified TPUs Some of these tasks proved to be feasible on implementations developed in previous work.

Independence from the monitor

As discussed in Section 4.3., it might be desired to remove the monitor PROM on the TPU in a final version of the TOS. Relevant initialization procedures which are provided by the monitor can be extracted and added to the Taskmaster. It is desired to organize necessary software and possible hardware modifications such that the monitor PROM can be replaced with an application RAM if desired. However, it should still be allowed to fully utilize the monitor if the corresponding PROM is returned.

Including of Relevant Monitor Features in the Taskmaster

The capabilities of the monitor can be scrutinized and those features which are useful for a Taskmaster system can be extracted. These features include the development, downloading, and some debugging capabilities. Instead of utilizing two separate PROMs for the Taskmaster system and evaluation tools, it should be determined if both features can be implemented in one single PROM. This should be definitely possible using a 12K PROM, and eventually using a 8 K PROM.

The IDLE State and low-priority Background Activities

When the task queue is empty and the TPU does not conduct communication, the TOS assumes the IDLE State and does not perform any application service. It would be interesting to examine to what extent low-priority background activities can be performed in this state instead of idling. Activities would qualify which are less important than those performed by Queue tasks.

An example of such an activity is the positioning of a complex robot arm. Hand and fingers of this robot arm can be positioned utilizing Queue tasks. Meanwhile, in an activity performed in the IDLE state, the whole arm is moved by a stepper motor. The arm is advanced whenever the attention of the system is not required for the queue tasks or communication. Finally, a second low-priority task queue might be implemented.

Medium Access Schemes

On a bus system, utilizing multiple units, data communication needs to be organized. This can be realized by specific medium-access schemes. Different schemes like carrier sense multiple access with collision detection (CSMA/CD) or Token-bus schemes can be examined [5,18].

Utilization of port P2 for a Terminal

Port P2 of the EVB which is exclusively used to connect a development system could be utilized to hook up a terminal. A setup should be developed which allows either the connection of a development system or a dumb terminal upon a particular hardware condition. This could elevate the distributed control system to a COLAN with some data exchange utility functions.

Setup of a Single-Chip Taskmaster System

The Taskmaster II system could be implemented as a single-chip based distributed system. All Taskmaster code could be burned on the MCU chip and a complete unit would then only include the MCU chip, some communication chips, and a RAM application memory.

Utilization of other other MCU68HC11 Family Members

Finally, capabilities of other MCUs of the MCU68HC11 Family could be examined and utilized. The on-chip EEPROM could be employed to store the address or other private data of TPUs. Furthermore, a table could be installed so that operational details of the Taskmaster could still be customized after a generic version of the TOS is provided.

A key, crucial for proper execution of the Taskmaster could be programmed on EEPROM to limit access of the system in single-chip mode only to eligible operators. The security feature of the MCU68HC11 could be utilized to protect software piracy in single-chip mode.

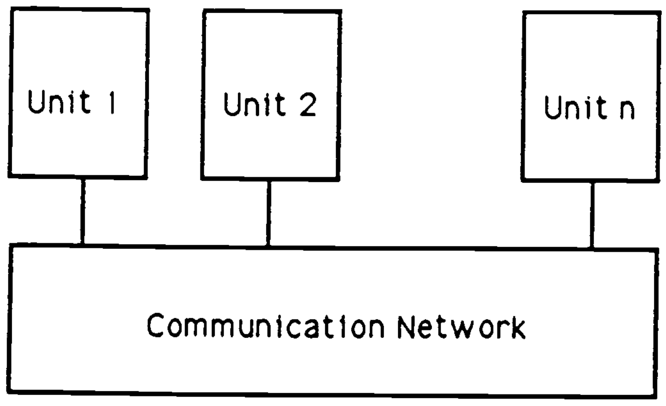

Figure 1-1

Distributed System

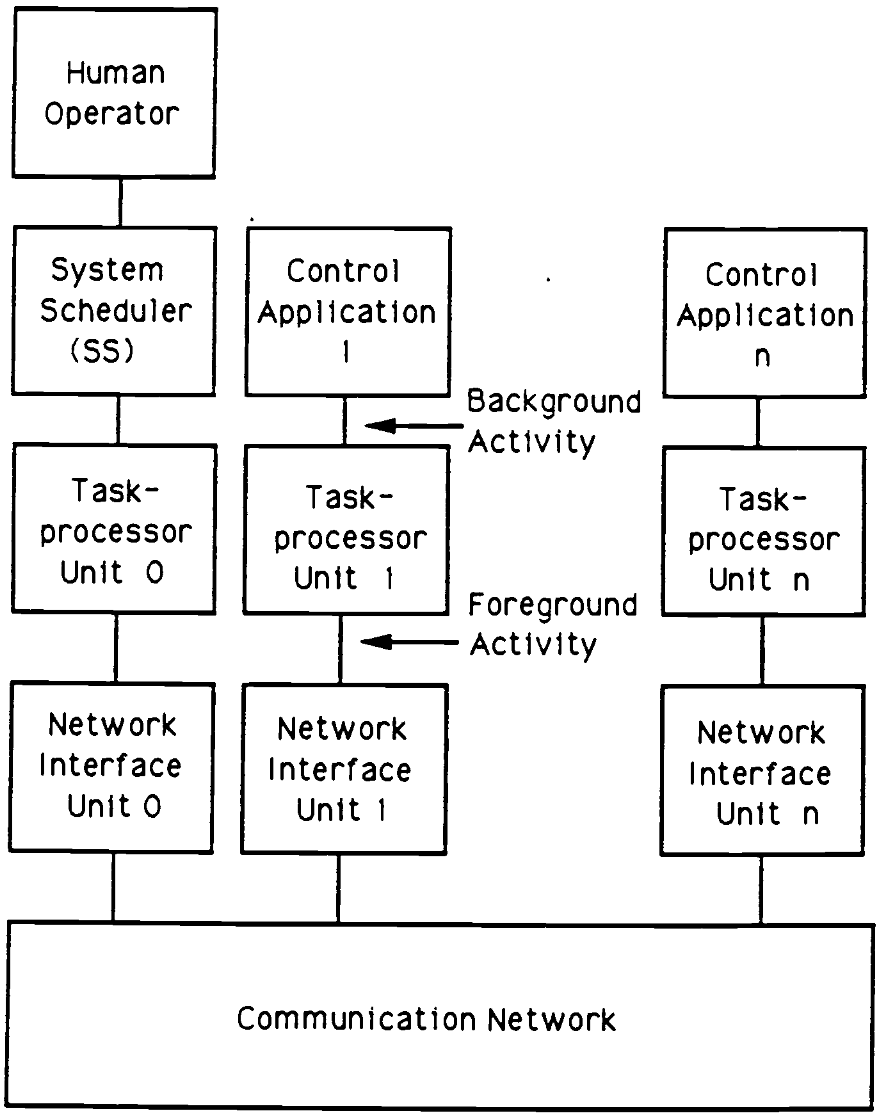

Figure 1-2

Taskmaster System

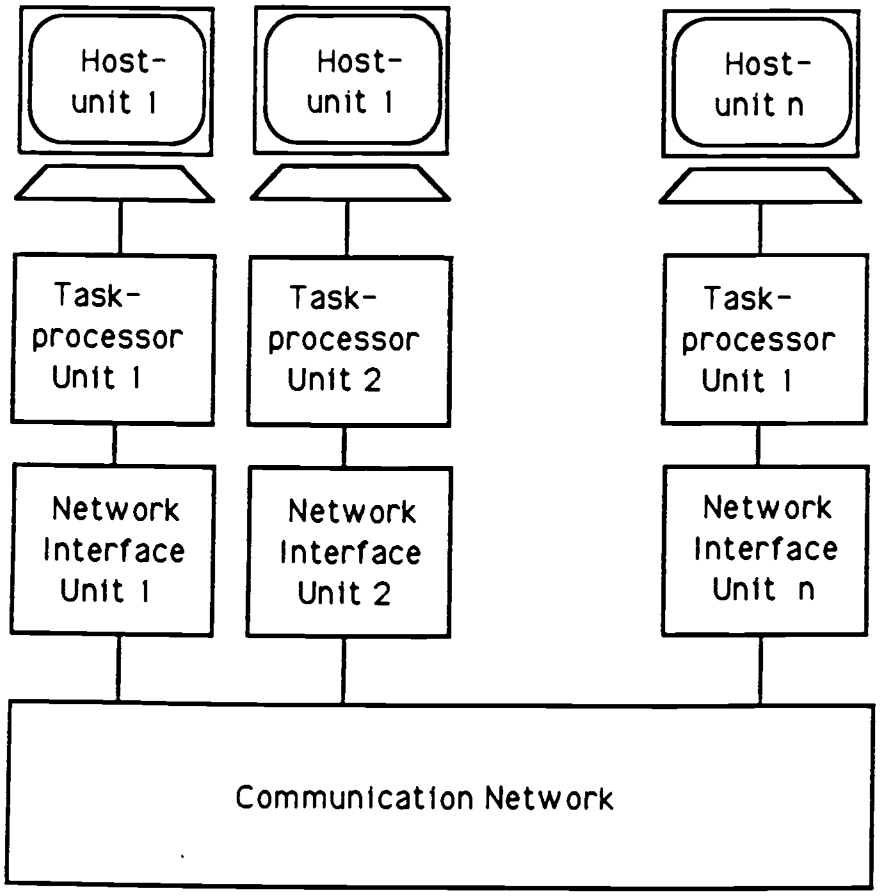

Figure 1-3

Taskmaster in a LAN Application

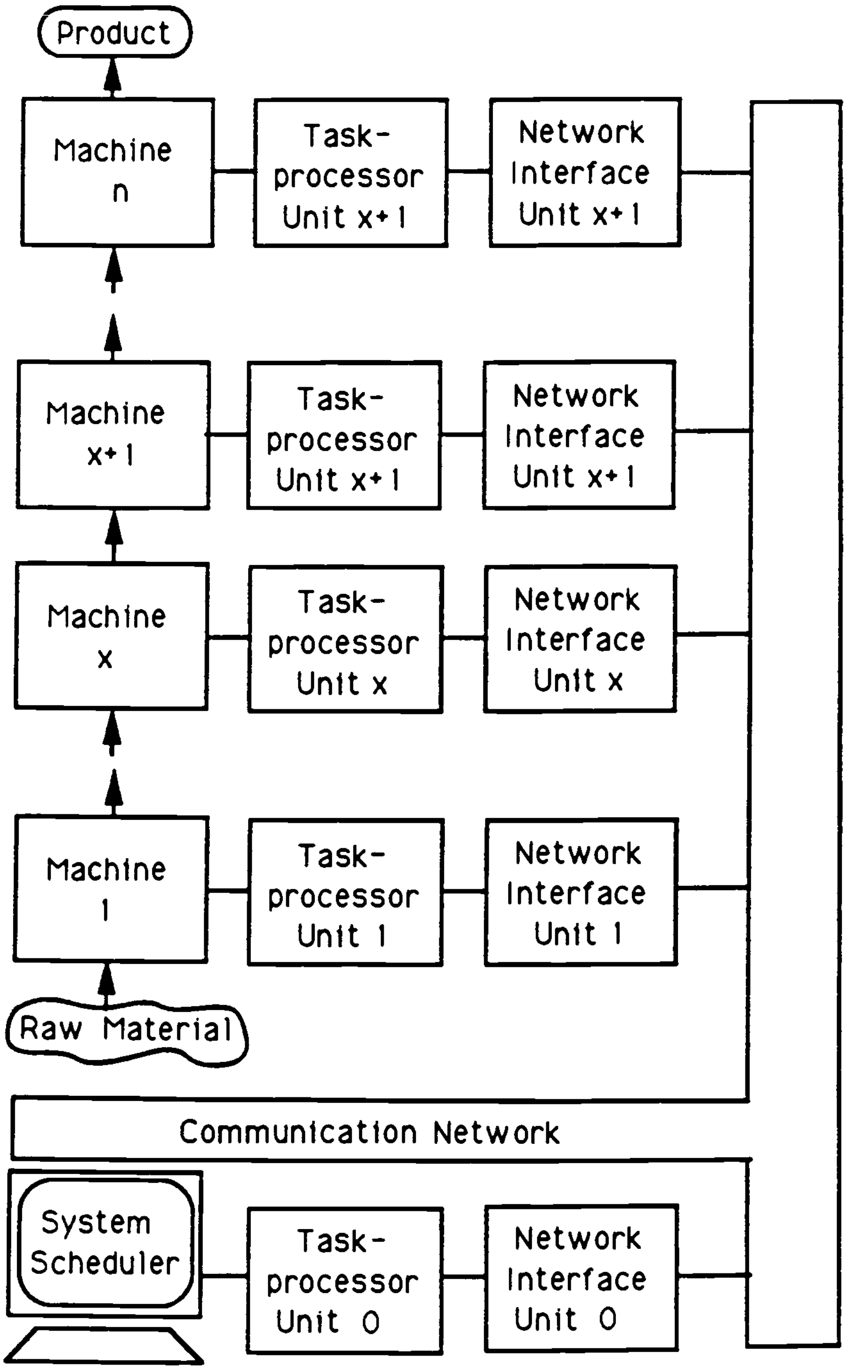

Figure 1-4

Taskmaster for Assembly Line Control

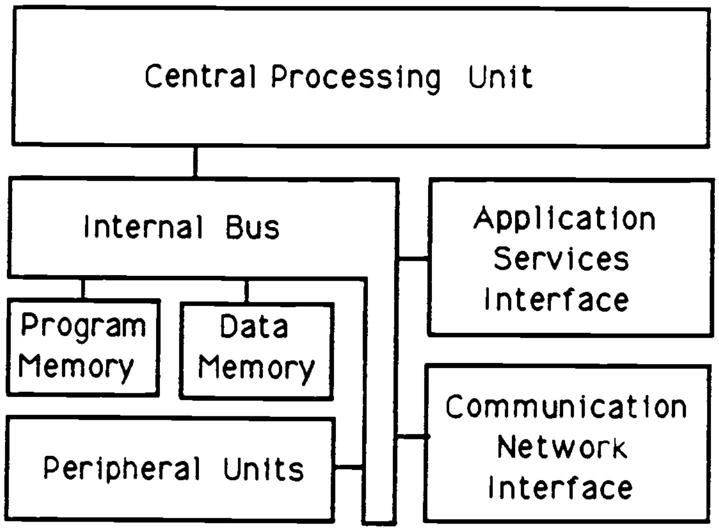

Figure 2-1

Morphology of a Taskprocessing Unit (TPU)

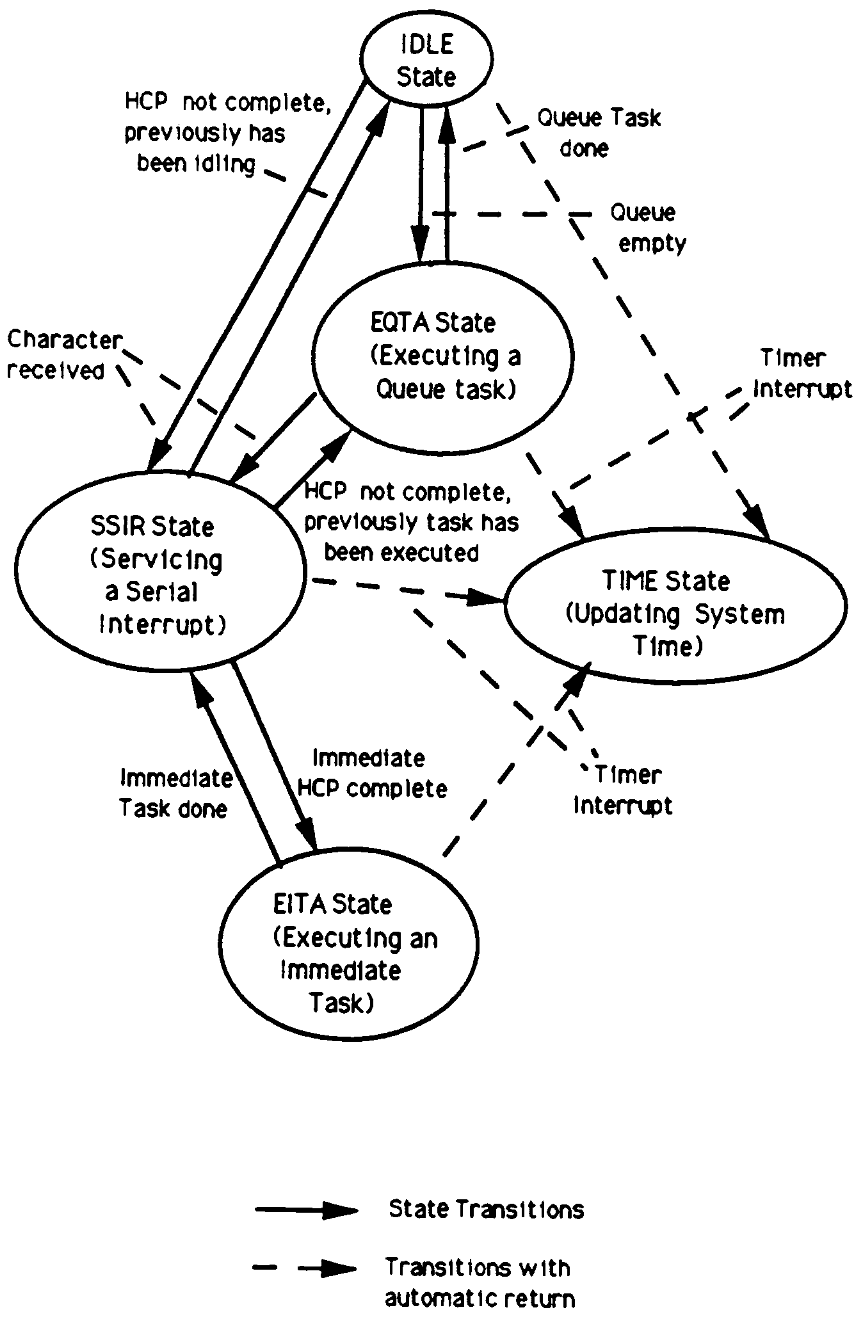
IDLE State
HCP not complete, previously has been idling
Queue Task done
Queue empty
EQTA State (Executing a Queue task)
Character received
Timer Interrupt
SSIR State (Servicing a Serial Interrupt)
HCP not complete, previously task has been executed
TIME State (Updating System Time)
Immediate Task done
Immediate HCP complete
Timer Interrupt
EITA State (Executing an Immediate Task)
State Transitions
Transitions with automatic return

Figure 2-2

State Description of the TOS

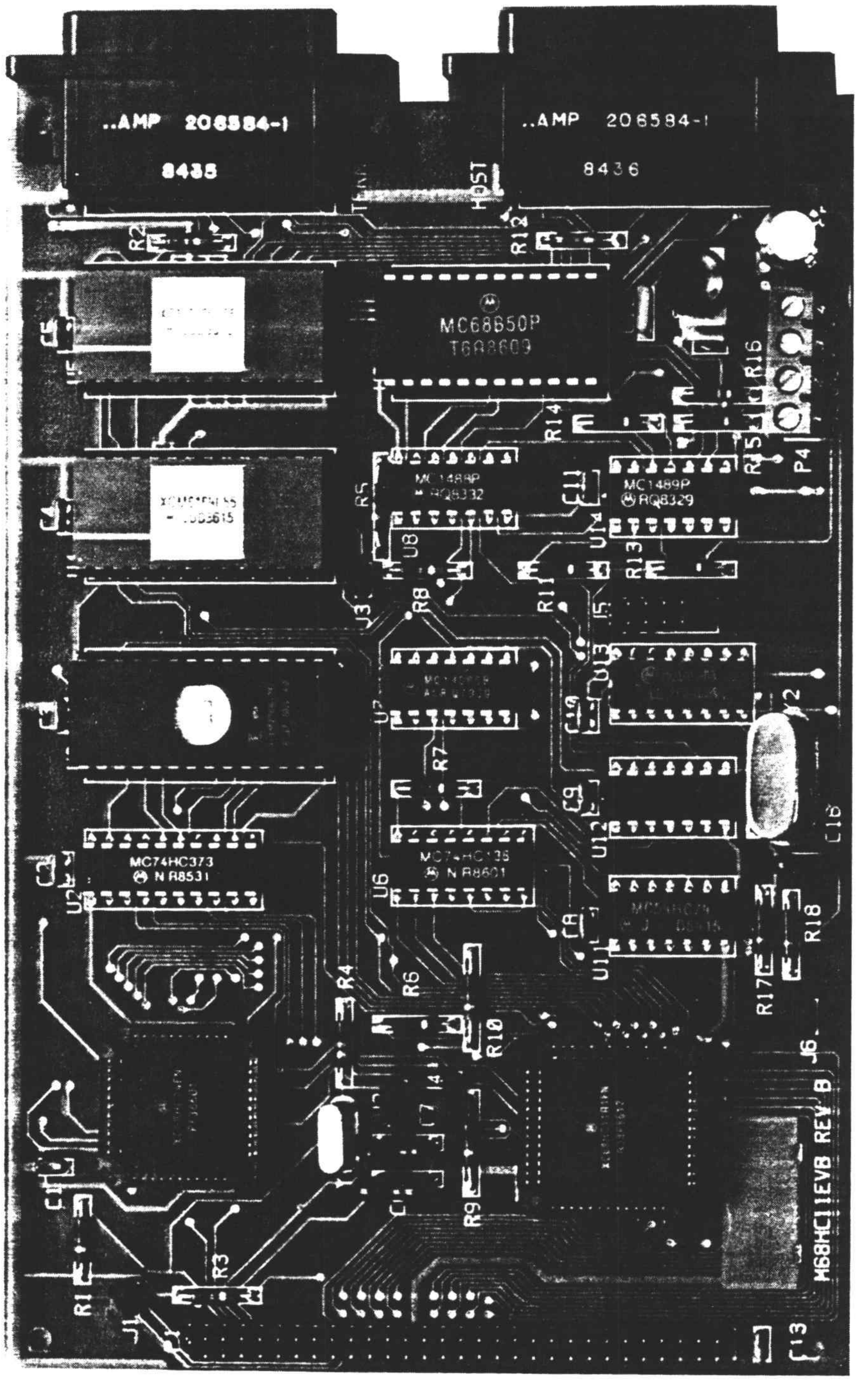

Figure 3-1

Evaluation Board

(reprinted with Permission from Motorola)

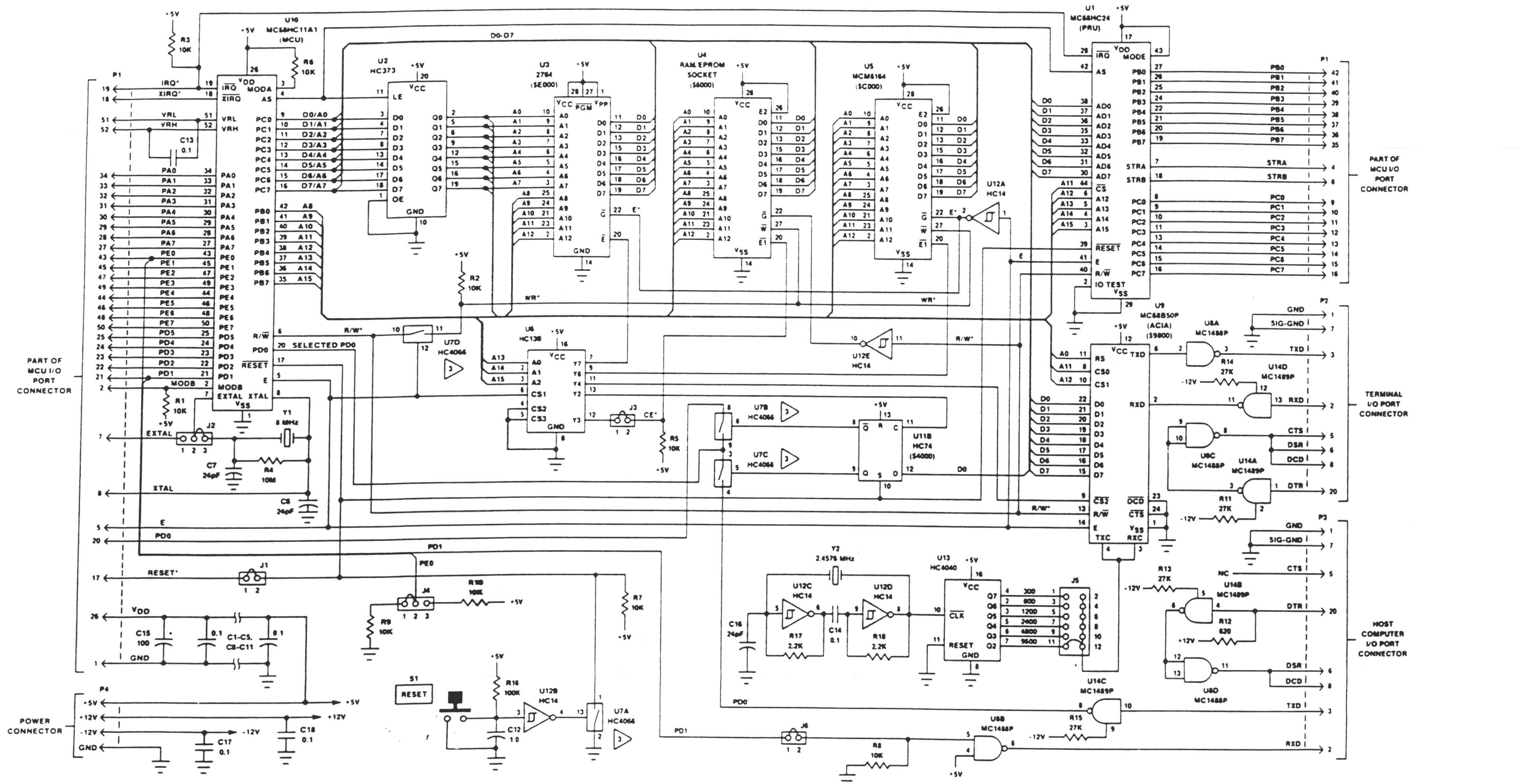

Figure 3-2

Schematic of the Evaluation Board

(reprinted with Permission from Motorola)

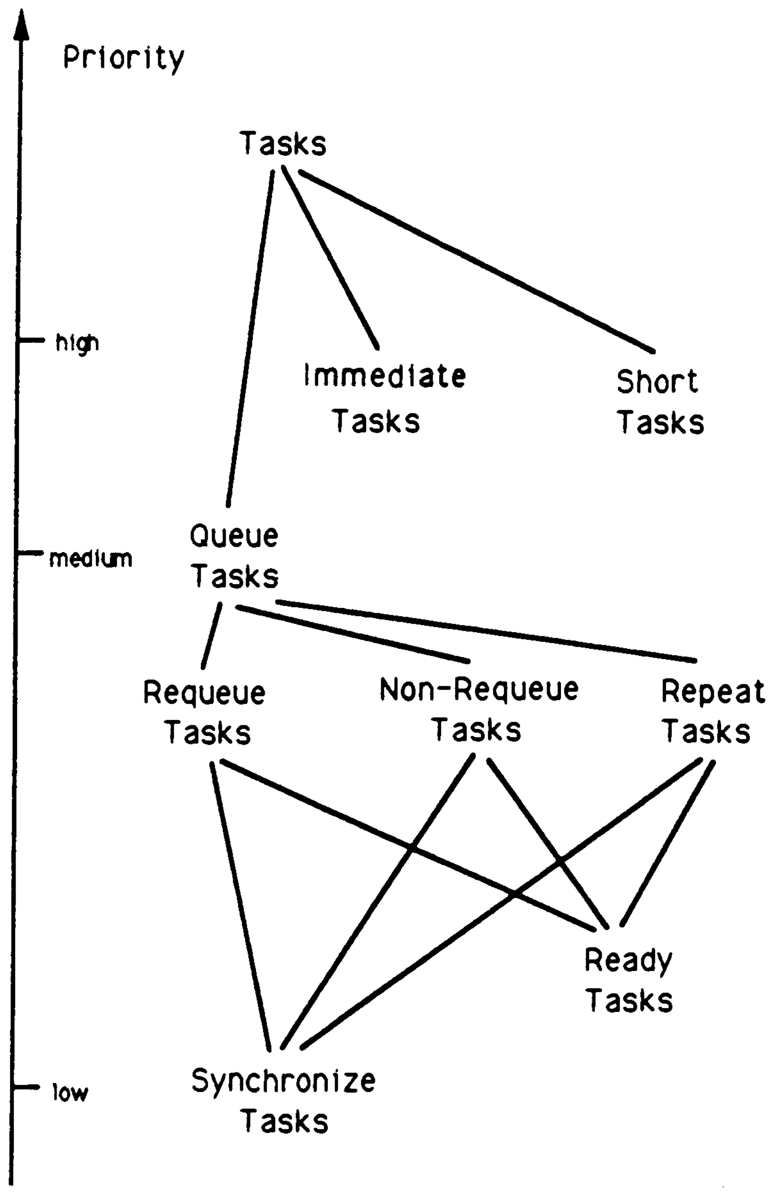

Figure 3-3

Linked Tree of Tasks

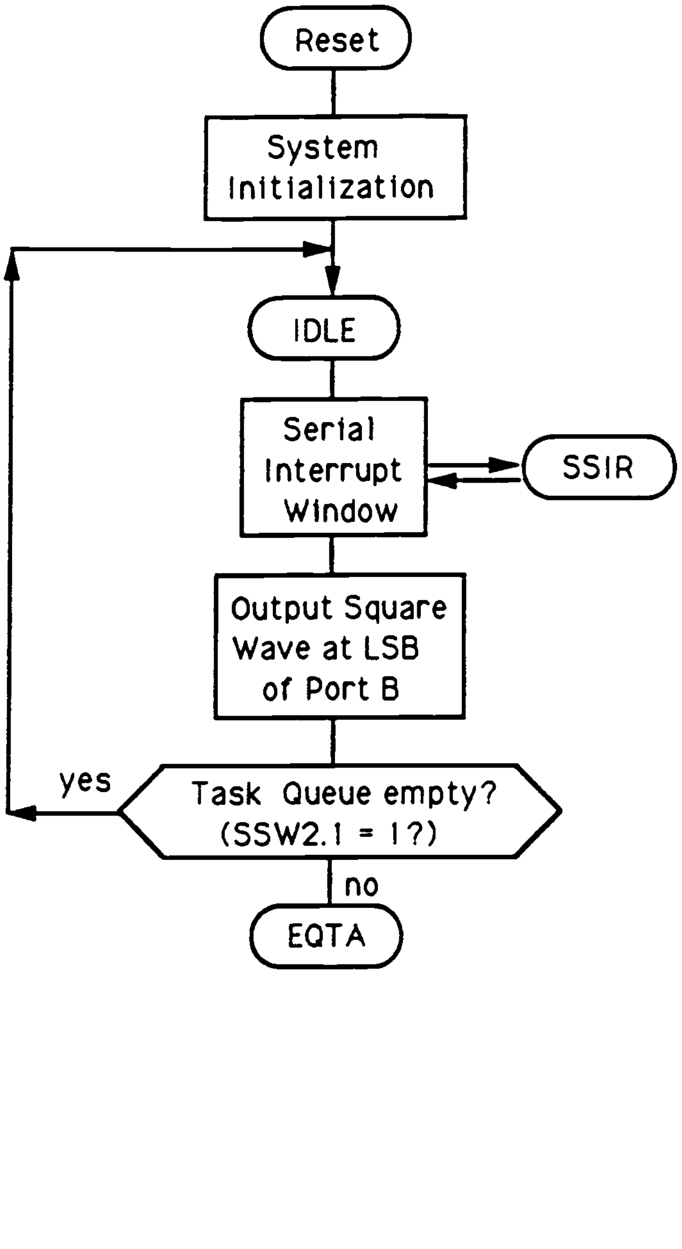
Reset
System Initialization
IDLE
Serial Interrupt Window
SSIR
Output Square Wave at LSB of Port B
Task Queue empty? (SSW2.1 = 1?)
yes
no
EQTA

Figure 3-4

IDLE State

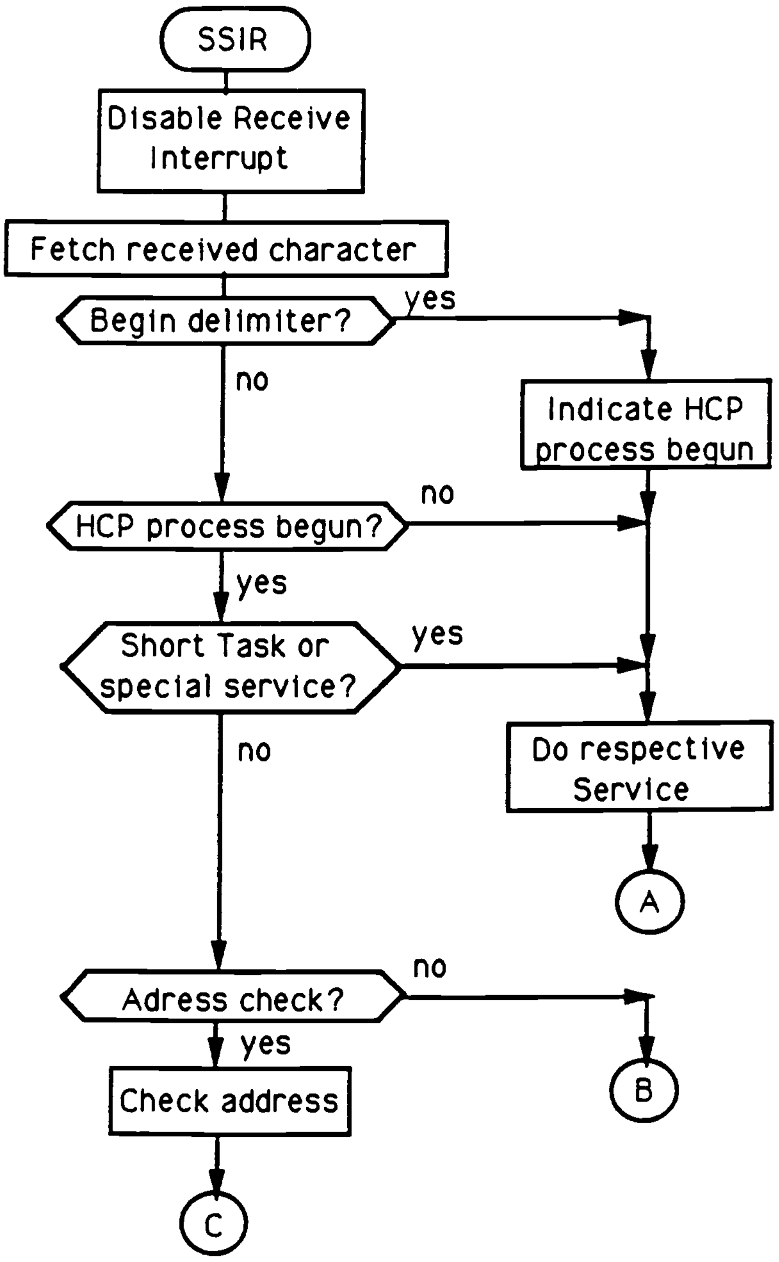

Figure 3-5

SSIR State

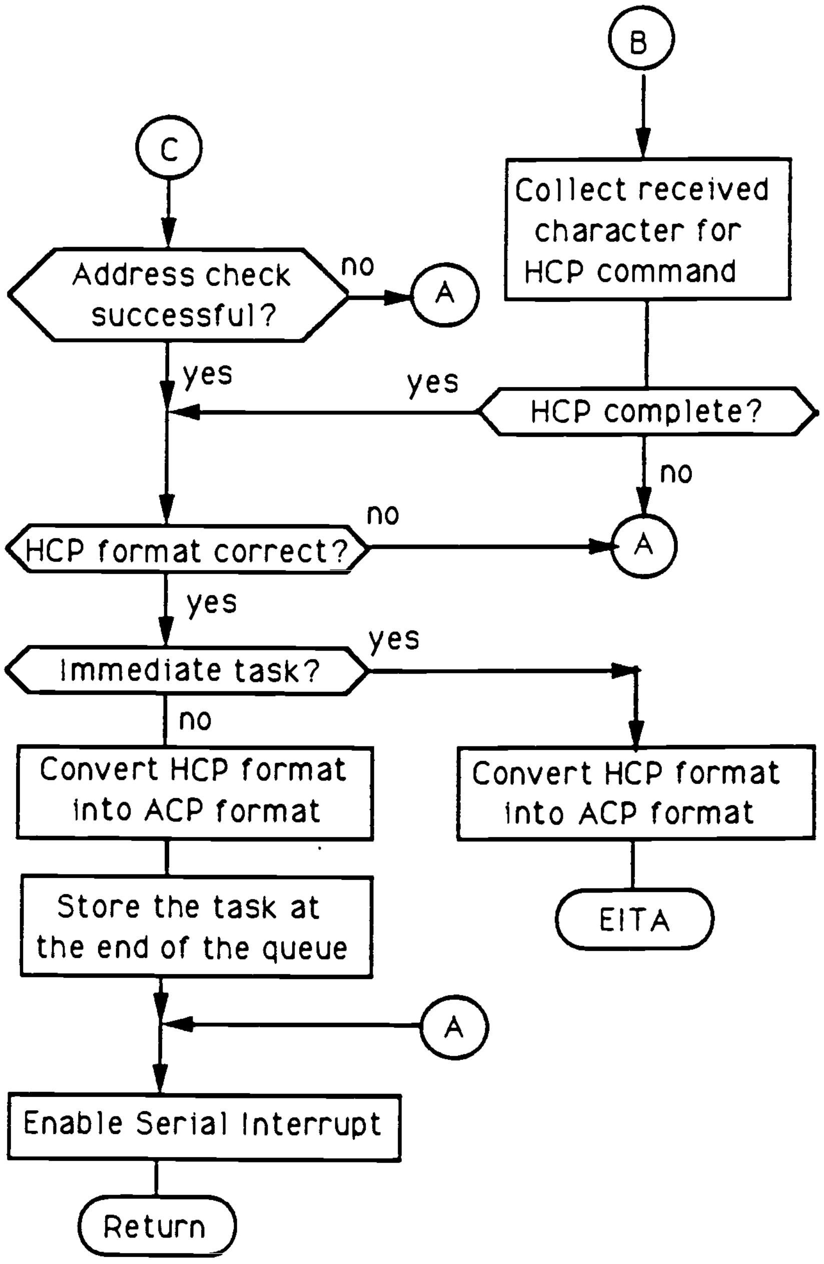
B
C
Collect received character for HCP command
Address check successful?
no
A
yes
yes
HCP complete?
no
no
HCP format correct?
A
yes
yes
Immediate task?
no
Convert HCP format into ACP format
Convert HCP format into ACP format
Store the task at the end of the queue
EITA
A
Enable Serial Interrupt
Return

Figure 3-5 cont.

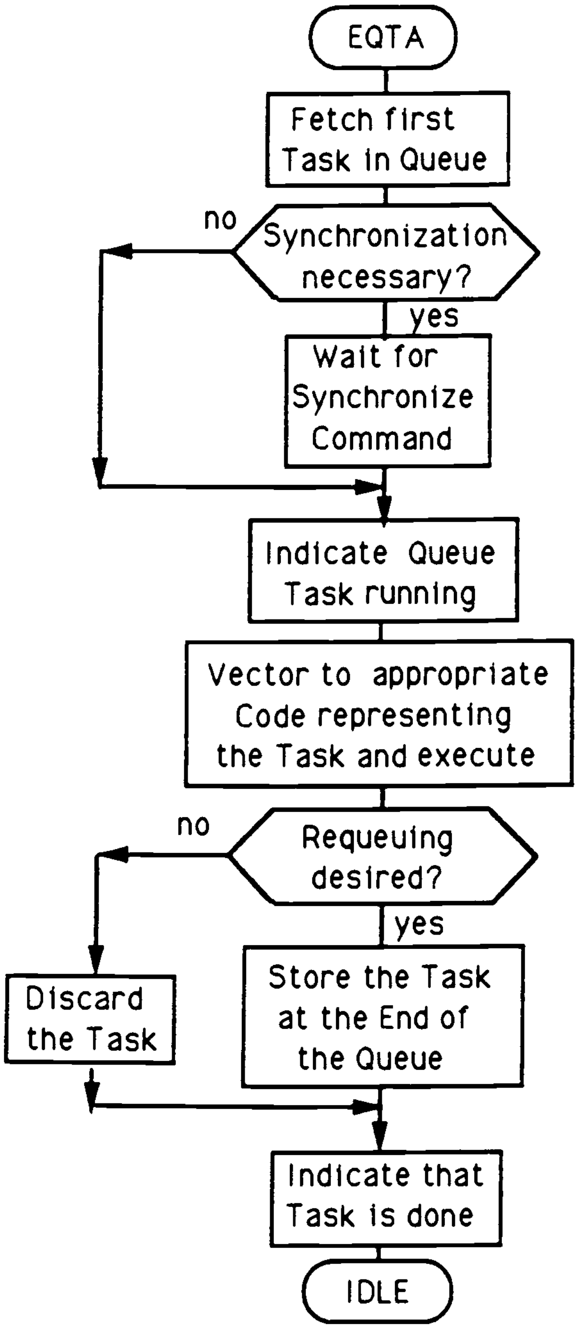
EQTA
Fetch first Task in Queue
Synchronization necessary?
no
yes
Wait for Synchronize Command
Indicate Queue Task running
Vector to appropriate Code representing the Task and execute
Requeuing desired?
no
yes
Discard the Task
Store the Task at the End of the Queue
Indicate that Task is done
IDLE

Figure 3-6

EQTA State

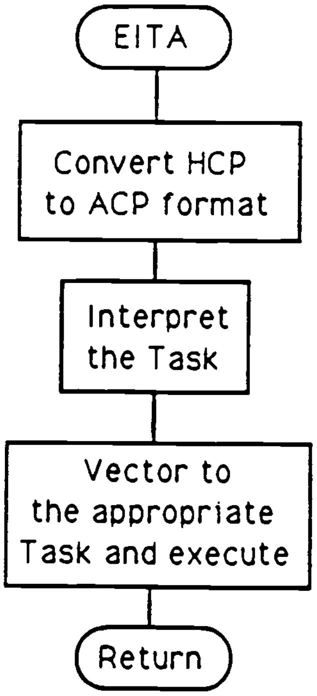
EITA
Convert HCP to ACP format
Interpret the Task
Vector to the appropriate Task and execute
Return

Figure 3-7

EITA State

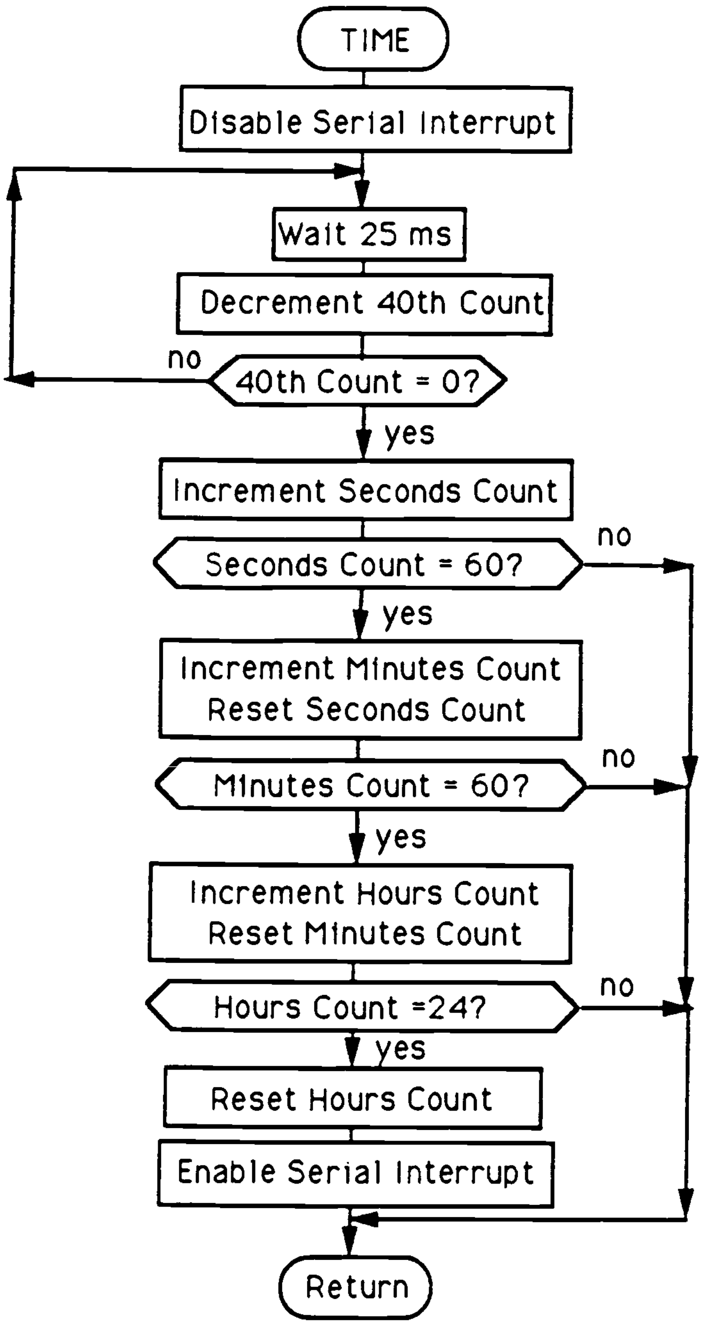

Figure 3-8

TIME State

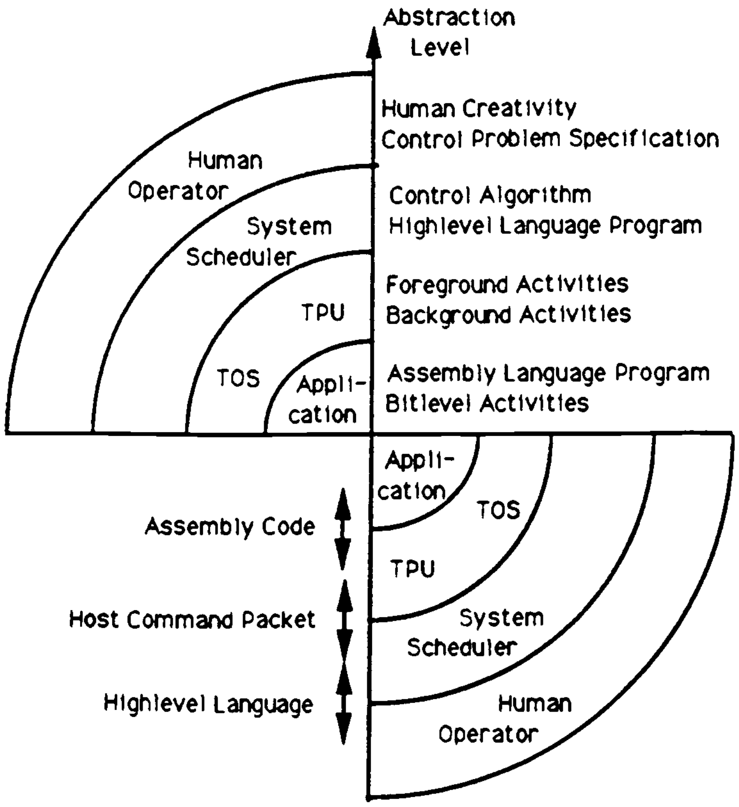

Figure 4-1

Layers and Levels of Abstraction

BIBLIOGRAPHY

[1] Brown, L. F., "A Role for Programmable Controlers in Factory Distributed Control", IEEE Transactions on Industry Applications, 1985

[2] Herzog, J. H., "A Design Perspective for Real-Time Control in Distributed Systems", IEEE Transactions on Industrial Electronics, February 1983

[3] Herzog, J. H., "A Design Methodology for Distributed Microprocessors in Real-time Control Applications". Paper presented at the Second International Conference on Computers and Applications, Beijing, People's Republic of China, June 1987.

[4] Hammer, M. "Distributed Systems, Past, Present, and Future", Computer Bulletins, July/August 1985.

[5] Reiss, L., "Introduction to Local Area Networks with Microcomputer Experiments", Prentice Hall Inc., Englewood Cliffs, NJ, , 1987

[6] Tannenbaum, A. S., Computer Networks, Prentice Hall Inc., Englewood Cliffs, NJ, 1981

[7] Georgopoulos, C. J., "Interface Fundamentals in Microprocessor Controlled Systems", Reidel Publishing Company, Dordrecht, Holland 1985

[8] Franta, W. R., "Local Area Networks", Lexington Books, Heath and Company, Lexington, MA, 1981

[9] Derfler, F. J. Jr., "A Manager's Guide to Local Area Networks", Prentice Hall, Inc., Englewood Cliffs, NJ, 1983

[10] National Computing Centre Limited, "Local Area Networks", NCC Publications, Manchester, M1 7ED, England, 1982

[11] Farowich, S. T., "Communication in the Technical Office", Spectrum, IEEE Inc, April 1986.

[12] Yu, Hua, "EDLAN: An Educational Local Area Network", unpublished Master's Thesis, Oregon State University, Corvallis, Oregon, March 1989

BIBLIOGRAPHY cont.

[13] Zhou, Yaqin, "ClASSLAN: An Experimental Personal Computer Network for Classroom Education", unpublished Master's Thesis, Oregon State University, Corvallis, Oregon, March 1989

[14] Zhen, Y., "A Simple Local Area Network, COLAN (Control Oriented Local Area Network), unpublished Master's Thesis, Oregon State University, Corvallis, Oregon, December 1986.

[15] Kao, S., "Design of COLAN II, A Control Oriented Local Area Network, unpublished Master's Thesis, Oregon State University, Corvallis, Oregon, September 1987.

[16] Eum, D. COLAN III, "A Control-Oriented LAN Using CSMA/CD Protocol", Unpublished Master's Thesis, Oregon State University, Corvallis, Oregon, September 1987

[17] Thye, Y., "COLAN IV, A Local Area Network for Communication and Control", unpublished Master's Thesis, Oregon State University, Corvallis, Oregon, February 1988

[18] IEEE Std. 802.3 Local Area Network Standard Carrier Sense Multiple Access with Collision Detection, IEEE Inc, 1985.

[19] IEEE Std. 802.4 Local Area Network Standard Token Passing Bus Access Method, IEEE Inc, 1985.

[20] Kohavi, Zvi, "Switching and Finite Automata Theory", McGraw-Hill Book Company, 1978

[21] Intel Corporation, 8031/8051 Microcontroller Handbook 1987.

[22] Motorola Corporation, "M68HC11 Evaluation Board User's Manual",1986

[23] Motorola Corporation, "M68HC11 Reference Manual", Prentice Hall, Englewood Cliffs, NJ, 1989