This is to certify that the

thesis entitled

HARDWARE-SOFTWARE CO-DESIGN FOR EMBEDDED SYSTEMS
IMPLEMENTATION OF A STEPPER-MOTOR CONTROLLER

presented by

Anuradha Mulukutla

has been accepted towards fulfillment
of the requirements for

Master's degree in Electrical Eng

David Fisher
Major professor

Date 12/17/98

HARDWARE-SOFTWARE CO-DESIGN FOR EMBEDDED SYSTEMS

IMPLEMENTATION OF A STEPPER-MOTOR CONTROLLER

By

Anuradha Mulukutla

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

1998

ABSTRACT

HARDWARE-SOFTWARE CO-DESIGN FOR EMBEDDED SYSTEMS

IMPLEMENTATION OF A STEPPER MOTOR CONTROLLER

By

Anuradha Mulukutla

Hardware-software co-design entails the combined specification of hardware and software at the system level, and the use of such a specification for co-simulation, co-synthesis and/or co-verification of the system. The co-design methodology is especially relevant to embedded systems which involve software with specific functionality embedded in microprocessors, often interacting with the environment and controlling external machinery. The objective of this thesis is to demonstrate the hardware-software co-design flow by using two implementations of a stepper-motor controller. One approach uses a hardware chip for the purpose, while the second approach is using the POLIS co-design environment and the PTOLEMY simulator. The results of the two implementations, and a study of hardware-software co-design considerations for system specification, architecture, hardware-software partitioning and the related trade-offs are presented, with specific reference to the stepper-motor controller experience.

DEDICATION

This work is dedicated to my parents.

ACKNOWLEDGEMENTS

This thesis would not have been possible without the guidance, encouragement, and support of Dr. P.D. Fisher. I extend my sincere thanks to him. I am also thankful to Ms. Roxanne Peacock and Mr. Brian Wright of the EE technical shop for their help. My special thanks are due to Mr. Robert Yu for his help with handy board and the introl software.

Contents

Chapter 1

INTRODUCTION

Conventional system design methods involve specification of hardware and software separately, requiring a pre-design partition into hardware and software. The integration of the hardware and software is pushed toward the downstream of system design. Since problems during testing or specification changes can only be corrected by expensive redesign, this leads to time-to-market and cost glitches. Hardware-software co-design tries to address such problems by the use of a combined or unified representation of the system, complemented by co-synthesis, co-verification and co-simulation of the system operation across the hardware and software boundaries. Co-design could address systems with mixed hardware-software specifications, or a general system specification. In any case, the co-design flow includes system level simulation, hardware-software performance and partitioning issues.

Typical application areas of embedded systems include consumer electronics,

telecommunications applications, automotive controllers, safety critical plant or medical instrumentation, listed in increasing order of criticality. All such systems contain a CPU running software with application-specific functionality, with interfaces to the environment, external hardware and in some cases other supervisory software on a real-time basis. These considerations make it necessary to deal with system level design issues rather than superior computing perfomance [1]; consequently, hardware-software co-design is the only alternative.

This thesis aims to understand the motivation for hardware-software co-design, explore available tools for co-design, demonstrate the co-design methodology for an embedded system application, and present the results of the implementation.

POLIS [2], a hardware-software co-design environment for control-dominated embedded systems, was selected as the design tool due to its relevance to the stepper-motor controller application, and its availability. The POLIS preferred specification language ESTEREL and the simulator PTOLEMY were used for the design. A software version of the stepper motor controller was generated using these tools and after synthesis and simulation, was tested on the 68HC11 microcontroller. Simultaneously, the hardware implementation of the stepper-motor controller using the MC33192 was set up and the results of the two approaches are presented.

1.1 Objectives

The main goal of this thesis was to demonstrate the hardware-software co-design methodology for a stepper-motor controller and establish a set of design criteria for system specification, architectural design, and hardware-software partitioning to facilitate the co-design process. Towards this end, some specific objectives were involved and these are briefly described below.

The relevance of hardware-software co-design for embedded systems was to be studied. Existing methodologies for hardware-software co-design were to be examined. A suitable co-design environment was to be chosen and used for the purpose of the implementation. The hardware implementation using MC33192 was to be realized and compared to the co-design implementation. The co-design flow used in the project was to be presented, along with insights gained into system design, hardware-software partitioning and typical trade-off issues. Current trends and issues in hardware software co-design were to be presented.

1.2 Outline

This thesis is organized into seven chapters, beginning with this introductory chapter. This chapter is followed by an overview of hardware software co-design in general and embedded systems in particular. The next chapter describes the co-design environment and design flow used for this project, namely POLIS and

its associated tools. The following chapter, chapter 3, deals with the software implementation of the stepper-motor controller using the co-design environment. The next chapter describes the hardware implementation, using the MC33192. Chapter six presents the results of both the approaches. The next chapter discusses the analysis and conclusions of the entire project while the final chapter presents some recommendations for future work.

Chapter 2

HARDWARE-SOFTWARE CO-DESIGN

2.1 A case for co-design

Traditional system design follows a sequential approach, called the "waterfall" model [9]. The design process begins with a system specification, (which may be simply the functional requirements of the system). Partitioning of hardware and software is done at this stage itself.

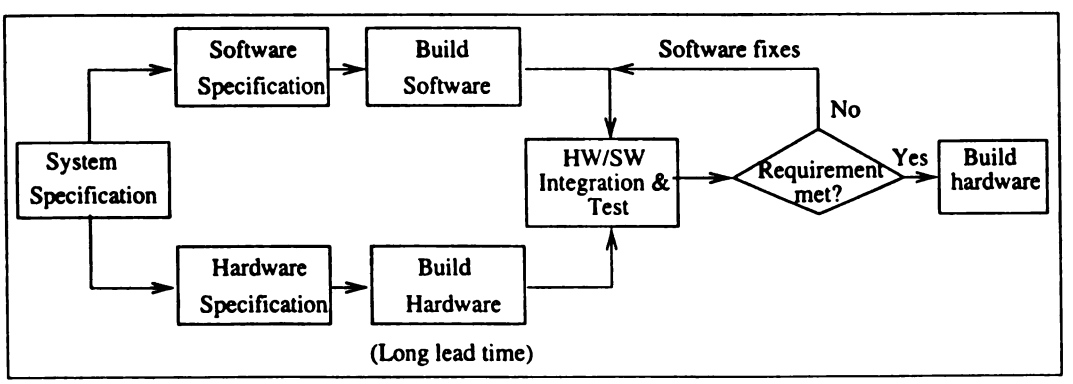


Figure 2.1: Traditional Method Of Design

Figure 2.1 [9], illustrates this approach. Designers try to map most of the specification to software, and use hardware only for timing and other constraints. Once this is done, as shown in the figure, the design process forks into two independent paths, one for hardware specification and design, and the other for software. This amounts to sending the hardware and software designers to separate work areas, with little interaction between them, until each team is ready with their product. Intuitively, this is a ludicrous beginning for the next design step, which involves integration and testing of the two products into the overall system.

Current-day digital applications involve increasingly complex systems, and contain both hardware and software components sometimes on a single chip. These components often include programmable microprocessors, ASICs and hard-wired devices or FPGAs. The market is highly competitive, ever increasing the pressure to decrease time and cost of product design. With this scenario, the traditional method suffers from obvious drawbacks, including the following.

- The pre-design partitioning of the system into hardware and software components prevents optimization of the design by exploring alternative partitions.
- Until integration and test, incompatibilities between the hardware and software portions cannot be found.
- Since hardware changes are not only expensive but also take time, only software alternatives may be explored to fix errors.
- Since problems are discovered late in the design phase and cannot be fixed completely merely by software modification, the resultant product is far from optimal and may not meet the specification.

In other words, we require a *concurrent* and not a *sequential* method of hardware-software design as shown in the Figure 2.2 [9]. This is commonly called co-design,

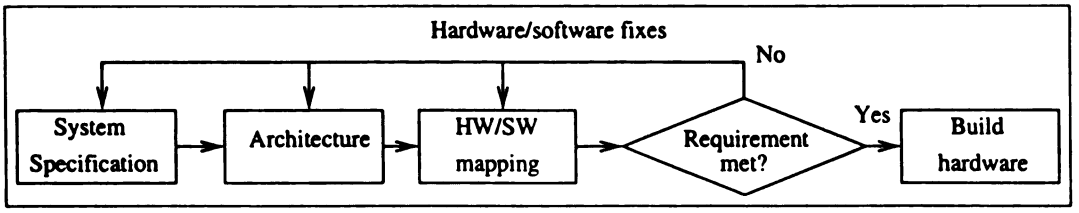


Figure 2.2: A Concurrent Method Of Design

and could be defined in many ways depending on the application. The key concept underlying these definitions is, however, "developing hardware and software concurrently"; (Hugo De Man in [11], Vijay Nagasamy in [10]) and "facilitating communication between hardware and software teams" (Karl Van Rompaey in [11]). This also means that hardware and software branches of specialization essentially overlap [1], and [12] and requires cultural shifts in design methodologies; however, we restrict ourselves here to the co-design methodology itself.

2.2 Methodology and Trends

Having decided that co-design is a very desirable methodology, we now describe a general approach for hardware-software co-design, which may vary, according to the application.

For this purpose, the co-design methodology is broadly decomposed into four major tasks [3]. These are, system specification, architectural design, hardware-software partitioning and iteration between hardware and software. The co-design process is complemented by co-simulation and co-verification which precede the

final implementation. An overview of these tasks is presented here for the general case. The next section deals with the specifics for embedded systems.

2.2.1 Specification

The functional requirements of the system are specified as the first step. The ideal case is to be able to arrive at architecture-independent specifications, for maximum flexibility during implementation. However, depending on hardware or software components involved in the system, the specification could deal with different levels of abstraction. The key is the unified specification of the entire system, using a suitable modelling mechanism. This is the subject of extensive research, and different techniques, models, and specification languages are continuously emerging both commercially and among research communities. Some of these are extended versions of C, extensions of HDLs [11], formal specification languages [8], [13] and [14] and recently, a unified co-design language or the CDL [15]. (Others are LOTOS and SDL for communication protocols, CSP and OCCAM for concurrent systems, StateCharts for real time systems, C, C++ and ADA for programming, SpecCharts for mixed hardware-software systems and so on).

2.2.2 Architecture

Architectural design may be influenced by the current application, available resources, or other constraints. The hardware architecture may contain one or more

processing units, memory subsystems, application-specific hardware blocks, (IP or intellectual property blocks) and other programmable hardware. The designer's job is to select an appropriate combination of such components, for the required functionality. Some tools which automate this process using cost and size parameters generated for the given functionality can be found in [16], [17], [19], [20], [21].

2.2.3 Partitioning

In general,the system specification is modelled as tasks or processes [19], interacting with each other. The partitioning step assigns each of these processes to hardware and software components available from the previous step. The main issues involved here are the granularity or abstraction of these processes, inter-process communication, and required concurrency. Different automatic partitioning algorithms [18], [20], [22] are available, which are based on cost, size or performance parameters estimated for the given hardware or software. In some cases, partitioning may be done manually, as in POLIS. The designer makes decisions based on these parameters considering timing, performance and hardware-software cross-coupling issues.

2.2.4 Iteration Between HW/SW

The hardware-software partitioning process is an iterative one, till an optimal design is achieved under the given set of constraints. Software offers flexibility, while hardware may be essential for standard functionality or timing constraints. Different hardware-software mappings of the system may give different results. The path from the co-design process to the final implementation of the system is supported by co-simulation and co-verification environments. These tools facilitate integrated simulation and verification of the hardware and software components of the system using various techniques.

Table 2.1: HW/SW Co-Design Tools - A Summary

Name	Category	Application	Platform	Source
POLIS [2]	Research Software	Design environment for control-dominated embedded systems	Unix	POLIS Project Cadence Berkeley Labs
Ptolemy [25]	Research Software	Extensible block diagram environment for signal processing, communications and and HW/SW co-design	Unix	Ptolemy Project UCB
Eaglei [28]	Commercial	Virtual Systems integration tool for co-verification in Embedded systems	Unix Windows NT	ViewLogic[1]
Statemate [14]	Commercial	Graphical Simulation and software synthesis tool for rapid development of embedded systems	Unix Windows NT	i-Logix Inc

[1] Formerly sourced from Eagle Design Automation Inc.

2.2.5 Co-Design Tools

While different tools are available with different capabilities to support co-design, these are domain-specific and often application-specific. Current research in the area encompasses different aspects of hardware-software co-design from specification languages to computational models to automatic partitioning tools. The Table 2.1 provides a brief look at some of the developmental and commercial tools available for co-design. Details relevant to embedded systems are presented in the following section of this chapter.

2.3 Embedded Systems

The co-design methodology outlined in the previous section is described in detail for embedded systems [3], [16]. An embedded system represents a reactive system with a fixed functionality and whose behaviour depends on its interaction with its environment. Often such systems could be very complex, comprising of software, ASICs, microprocessors, FPGAs and analog peripherals. Co-design allows concurrent development of hardware and software, offering great reductions of time-to-market compared to software development after hardware fabrication.

We discuss the co-design tasks from the previous section, as applied to embedded system design using the example of an engine-control unit hereafter referred to as ECU, from [24].

2.3.1 Specification

Initially, the functionality of the system is specified. This is a set of system requirements or the response of the system to inputs. In case of the ECU, the function is to control the torque produced by the engine by timing the fuel injection and spark, with low fuel consumption and low exhaust emission. The injection time is computed by using air pressure, air temperature, throttle position, engine speed etc. The ECU produces a suitable output to drive the actuators based on this computation. We can thus divide the basic functionality into subtasks. This functionality is mapped into modules by using a conceptual model [16]. This may be done using various models like data flow graphs or finite state machines(FSMs). A description of this model is then generated by means of a specification language. At this point, the specification does not reflect any implementation detail.

The choice of a model depends on the application. For instance, for a signal processing application, a dataflow model might be suitable. The FSM model may be more applicable to control-dominated systems. Software systems may need to be modelled as communicating sequential processes or CSPs [16]. In short, the model should be most appropriate for the characteristics of the system. While embedded system applications are diverse, certain characteristics [1], [16] are common among such systems.

- Software with application-specific functionality
- State transitions responding to inputs
- Exceptions for interaction with environment or subsystems

- Concurrency

These characteristics require a FSM based model, extended into a model which also provides for concurrency. [16] describes one such model, the PSM or a program state machine model. The specification language Esterel, used for this thesis uses a model based on the CFSM or the co-design FSM, described in the next chapter. Once a description is generated, the specification is verified for functional correctness.

2.3.2 Architecture

Typical embedded systems have complex architectures, consisting of different types of processors and their peripherals, FPGAs, ASICs, and external sensors or electro-mechanical devices. Often, the design is incremental, using standard hardware parts or reusable software [5]. The selection may be between 16-bit and 32-bit processors, interconnection schemes like buses, or available alternatives for custom hardware. The criteria could involve both technical and commercial trade-offs.

In our ECU example, we may use a 32-bit CPU which receives the analog and digital inputs from the environment, and directly produces actuations in the form of PWM outputs. While this may be an easy approach, timing requirements may not be met. Alternatively, we might use a 16-bit CPU and an FPGA which produces the actuations. The third option may be to use a DSP to process inputs,

and an 8-bit CPU which computes the outputs and and FPGA which controls the actuators [24].

2.3.3 Partitioning

As mentioned previously, the partitioning task involves the allocation of architectural components to the functional operations. At this stage, co-simulation of the partitioned design can help in performance analysis. Thus, if timing information for a processor is available, then, the performance analysis can indicate whether the design can meet all the system requirements. The advantage of co-design is that flaws can be detected at this stage itself, and necessary re-partitioning or even re-design can be done.

Once again coming back to our ECU, partitioning may be dictated by one of several criteria. For instance if a variable is defined in the DSP, while a function using that variable is defined in the CPU, we need to design a suitable interface or bus for this purpose. Instead, it might be easier to define both the variable and the function in the CPU. Another example would be whether we assign some of the data processing functions to the CPU or use it only for computation. This choice may be dictated by the processing power available or the speed of computation required. As described in the previous section, this process is iterative. The design is then synthesized to generate software code and hardware netlists. The final stage of the design is the physical implementation using prototyping and

testing.

2.4 Summary

In this chapter, several important issues concerning hardware-software co-design have been discussed. The need for co-design is established. A general methodology for co-design has been described. A brief overview of existing tools and methods is presented. We then discussed the co-design methodology specifically for embedded systems.

However, hardware-software co-design is also riddled with some inadequacies. The foremost of these, is the fact that co-design techniques tend to be application-specific and diverse. No industry standard has emerged yet and the field abounds with specification languages, co-simulation and co-synthesis environments and verification methods. In other words, there is no universal solution [3]. This presents a challenge for designers both in terms of the choices to make, and also the required learning curve for changing technologies.

For this thesis, we have chosen the POLIS co-design methodology, due to its suitability for control-dominated applications. In addition, the POLIS system coupled with the PTOLEMY simulation environment provides a platform for system level design right from specification to final implementation. In the next chapter, we describe each component of the POLIS co-design environment, for performing each of the co-design tasks outlined here.

Chapter 3

THE POLIS SYSTEM

3.1 Introduction

POLIS is a co-design environment for control-dominated embedded systems and has been freely available on the internet since 1996. The software was created after almost a decade of combined effort by Magneti Marelli, a major European producer of automotive electronics, the University of California Berkeley, and many others [2]. The motivation for the POLIS project was to find solutions for challenges facing the embedded system industry, some of which are listed here [5].

- Formal customer specifications allowing changes during design
- Use of high level languages for software design
- Hardware-software co-design and co-simulation instead of bread-boarding for verification
- Design reuse

Embedded systems have to deal with changing product specifications, and simultaneously attempt to lower the design costs, while trying to reduce the time-to-market. In order to achieve these objectives and address the issues listed above, the POLIS project was conceived and implemented.

This chapter describes the POLIS co-design environment, including the Esterel specification language and the Ptolemy simulation environment. We begin with a description of the formal model used in POLIS, for system specification.

3.1.1 Formal Model

As discussed in the previous chapter, we require a formal model to specify the system. This model is called the CFSM or the Co-Design Finite State Machine. It is based on extended finite state machines, with a finite set of variables. Each functional module of the system is mapped to a CFSM. The CFSM specification is implementation-independent, and could represent either hardware or software components. Figure 3.1,shows the CFSM specification of a simple example taken from the POLIS users manual [4]. The operation of the CFSM is described later in this section.

In a CFSM, the states of the internal variables as well as the outputs, are updated by state transitions. The result of the transition is propagated to other CFSMs or the environment. The communication between CFSMs is not by shared variables, but by events. The authors of POLIS [5] call the model "globally asyn-

chronous and locally synchronous." This characteristic allows for the specification of systems consisting of hardware and software components, which exhibit asynchronous communication.

POLIS uses an intermediate language called SHIFT, (**S**oftware **H**ardware **I**nterchange **F**orma**T**) to represent CFSM networks and the individual CFSM behavior. The designer can specify the interconnecting netlist between CFSMs graphically by using Ptolemy or by a textual netlist file.

Events are emitted by CFSMs or the environment by means of data or control carriers called signals. The events are detected by one or more CFSMs. As events are not buffered, the designer needs to take explicit measures in order that events are not overwritten if transmitting and receiving CFSMs have different speeds. These can be handshaking mechanisms, partitioning and scheduling choices.

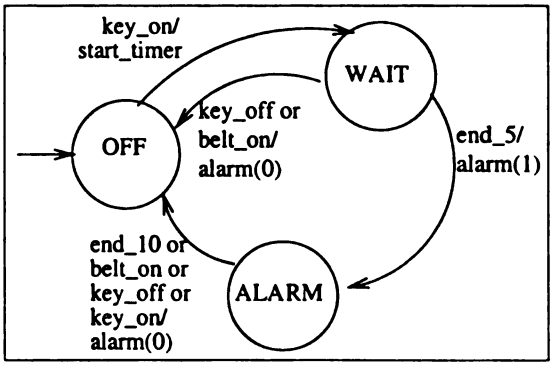


Figure 3.1: CFSM For Seat Belt Alarm

The Figure 3.1 shows the CFSM specification for an automobile seat belt alarm function. Five seconds after the key is on, an alarm is sounded if the seat belt is not fastened. The alarm beeps for 5 seconds or until the key is off or belt

is on. The transition labels use the "/" to separate the stimulus and reaction. This specification can be expressed in a high level specification language with CFSM semantics, in this case Esterel. The Esterel specification for this CFSM is described later in this chapter. The next section describes each step of the POLIS design flow.

3.1.2 The POLIS Design Flow

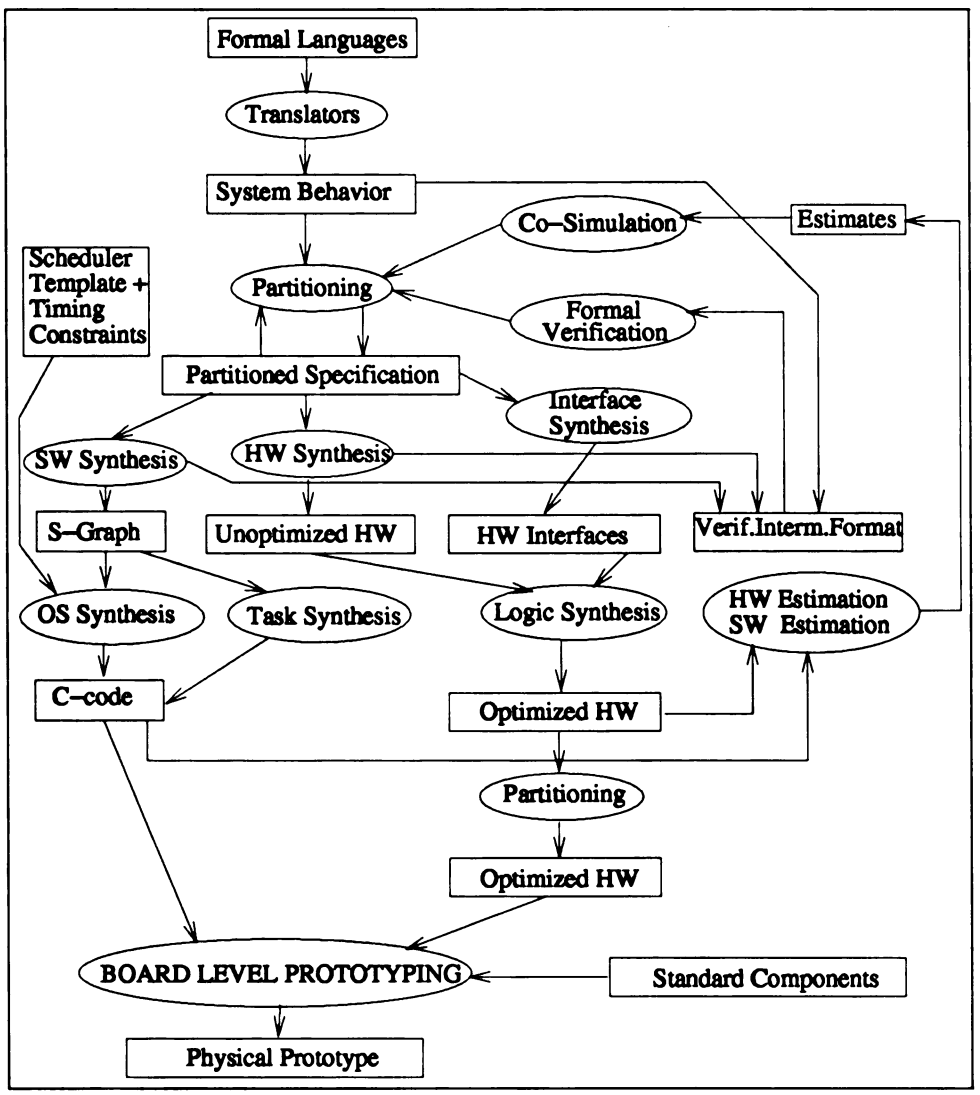


Figure 3.2: The POLIS Design Flow

The Figure 3.2 from [4] depicts the various design steps involved in the POLIS methodology. These individual steps are described in the following paragraphs. We begin with a formal specification of the system.

3.1.2.1 Formal Specification

Designers specify the system using a high level language with extended FSM semantics, in this case, Esterel. (Other such languages could be StateCharts [6], and the so called synthesizable subsets of VHDL or Verilog.) An example of such a specification is described later in the section on the Esterel language.

3.1.2.2 Translation

The Esterel files with extension .strl are translated to SHIFT format using the strl2shift translator. The input files describe the CFSM behavior and auxiliary information like type and constant definitions. The output contains the .shift files for the entire system and .cfsm files for individual CFSMs. The SHIFT files are hierarchical netlists containing blocks which can be CFSMs, functions or other netlists, and describe the signals between communicating CFSMs. More information about SHIFT can be found in [29] and [30].

3.1.2.3 Co-Simulation

The generated files are input to POLIS which creates synthesized C code to model all the system components, independent of their final implementation. (The POLIS design flow is managed using UNIX makefiles, allowing ease of use.) The co-simulation framework uses this C code as the basis. The Ptolemy simulator is used for this project. (A VHDL simulator could also be used [4].) Initially, a func-

tional simulation is performed to find and fix bugs. Then, a clocked simulation can be run with approximate timing. The co-simulation depends on the mappings obtained by the software synthesis step, which is done after partitioning.

3.1.2.4 Partitioning

The next step involves partitioning the design, i.e., mapping individual components, or CFSMs, to hardware or software. This can be iterative and uses the same user interface as co-simulation. Any number of alternative partitions can be tested.

3.1.2.5 Software Synthesis

The software CFSMs are mapped to software structures including CFSM procedures and an RTOS(Real-Time Operating System). The synthesis is performed in two steps.

- First, the CFSM behavior is represented using an S-graph or software-graph, which is similar to a control/data flow graph.
- Then, the S-graph is translated into portable C-code.

This C-code is optimized in a specific micro-controller-dependent instruction set using a suitable compiler. In addition, a timing estimator provides estimates of program execution times on a selected target processor.

The RTOS is application-specific, with user-selected scheduling algorithms. Hardware-software interfaces in the design are automatically synthesised as part of the RTOS by POLIS.

3.1.2.6 Hardware Synthesis and Prototyping

CFSMs selected for hardware implementation are mapped into abstract hardware description formats like BLIF(Berkeley Language Interchange Format), VHDL, or XNF (for implementation on FPGAs). A physical prototype of the system can be obtained using the .xnf files and Xilinx FPGAs. Since a software version is implemented for this thesis along with a hardware implementation using an existing chip for a stepper-motor controller, this facility is not used for this thesis.

3.1.2.7 Other Capabilities

- POLIS also facilitates formal verification through a translator from CFSM networks to synchronous classical FSM networks, for input to formal verification algorithms. The system recommended by POLIS is VIS or Verification Interacting with Synthesis [7], from the University of California at Berkeley. This project does not include formal verification.
- POLIS offers some support or use of micro-controller peripherals like timers and A/D converters. The currently supported micro-controllers are the Motorola 68HC11E9 and 68HC11gauss.

3.1.3 System Requirements For POLIS

Table 3.1: System Requirements

Package	Availability	Disk space
POLIS 0.3 [2]	Sun OS4 Solaris 2 DEC Alpha PC Linux	20-50Mb
Ptolemy 0.7 [25]	Solaris 2.5 HP-UX and others Linux[1] (with X-windows)	350Mb
Esterel v5 [8]	Sun solaris DEC Sun OS4 IBM AIX	25-30Mb

Table 3.1 summarizes the system requirements for the POLIS system including Esterel and Ptolemy. For this project, all the software was installed for Sun solaris 2.5. All the above packages can be freely downloaded from the world-wide web. In the next section, the Esterel specification language is described and the CFSM example mentioned before, is discussed in detail.

[1]Binaries contributed by others

3.2 ESTEREL

3.2.1 Introduction

Embedded systems could be classified under a class of computerized systems called *"reactive systems"*. They continuously react to external stimuli from the environment and their response is mainly input-driven. Further, the output values could be continuously produced from the inputs, as in signal processing applications. This is called "*data handling*". On the other hand, producing discrete output signals from input signals is called *"control handling"*. The stepper-motor controller system falls under the category of control-dominated systems.

Esterel [8] is a specification language aimed at the control-dominated components of reactive systems. Esterel affords a concurrent programming environment required for reactive systems, as they interact concurrently with the environment and are often made of concurrent modules communicating with each other.

3.2.2 Language Features

In a sequential program, an output is produced after some computation using the input data. The programmer specifies the order in which the program statements are executed. However, this model is inadequate for reactive systems, which exhibit real-time interaction with the environment, where input/output sequencing is important.

Esterel is based on a synchronous model, which accomodates this requirement, where the program reacts instantly or synchronously with the input. This model results in a distinctive language style for Esterel, involving timing concepts, and a deterministic behavior. (A deterministic program produces the same output given a particular input any number of times.)

In Esterel, signals are called events, which can be emitted and detected. An output event is the status of an output, computed from a given input event, which is the status of the input. These events are communicated in Esterel, by *"broadcasting."* This means, in a system with several modules, the emission of any event is available to any module which is interested in that event. The emitting module need not have information about the receivers. Thus, an event need not be replicated in the system. This *"broadcast semantics"* distinguishes Esterel from languages like VHDL.

Although Esterel concurrent modules are synchronous, as POLIS semantics are globally asynchronous, there is a difference between the two. In POLIS, synchronous behavior is available only till the boundary of a single CFSM. Composition of CFSMs is asynchronous.[1]

The basic programming unit in Esterel [27] is a module, with an interface declaration and a body. The language has a host of useful constructs including signal handling, looping, control statements, parallel constructs, and a wide variety of temporal constructs. Esterel supports external functions in two host languages,

namely C and Ada. C functions were used for this thesis. Since it is not possible to treat the entire language syntax here, the following example from [4] is used to give an introductory idea to Esterel. The module listed here, essentially performs the seat-belt alarm function, which was represented by a CFSM in the previous section. The end_5 and end_10 signals are received from a timer module and indicate end of the 5 second and the 10 second intervals.

```
module belt_control:
input reset, key_on, key_off, belt_on, end_5, end_10;
output alarm: boolean, start_timer;
loop
  abort
    emit alarm(false)
    every key_on do
      abort
        emit start_timer
        await end_5;
        emit alarm(true);
        await end_10;
      when[key_off or belt_on];
      emit alarm(false);
    end every
  when reset
end loop
```

The specification consists of an interface declaration and the body of the module. The interface consists of inputs, outputs, constants and any external functions. In this example, all the inputs except alarm are *pure signals*, *i.e.*, they do not have an associated value. The alarm signal has an associated true or false value. This module is executed as a continuous loop which is restarted whenever a reset signal is received. The *await* construct is one of the many temporal

[1] As a result of this, certain features of Esterel cannot be used in POLIS. Some of these are discussed in later chapters.

constructs offered by Esterel.

3.3 PTOLEMY

3.3.1 Introduction

Ptolemy [25] is a software environment supporting the design of reactive systems using heterogeneous modelling, and was developed at the University of California, Berkeley. Ptolemy provides support for heterogeneous prototyping of systems in areas like signal processing, communications and real-time control applications among others. The key principle is the use of mixed models of computation, realized by a specialized design style called *domain*, including synchronous data-flow (SDF), dynamic data-flow (DDF) and discrete event (DE) models, the first two being used mainly for signal processing, and the last for communications and control applications [26]. Some other domains are also supported. These domains could be used for simulation or code generation. We first describe the Ptolemy user-interface and briefly discuss the *DE domain*, which is used in POLIS for co-simulation.[1]

[1] Apart from Ptolemy, the POLIS documentation describes the use of commercial VHDL simulators or software simulation [4]

3.3.1.1 The Graphical Interface

The Ptolemy interactive graphical interface or *pigi* is a design editor for Ptolemy, and allows the graphical construction of designs by connecting icons. This is based on *vem*, a graphical editor for *oct*, which is the Ptolemy design manager.

Each domain consists of a set of *blocks*, *targets* and associated *schedulers*. A design is represented as a network of blocks. These blocks can communicate through *portholes* [26]. These blocks could be hierarchical, called *stars*, *galaxies* and *universes* from the lowest to the highest level. A *target* manages the simulation or execution of the block, and is derived from the block. The simulation performed by the target is governed by a *scheduler* which controls the sequencing of the execution of functional modules in the design.

The *pigi* editor provides palettes containing icons for design blocks and enables the user to create a graphical netlist or schematic for a particular design using the icons. A target is generated for each design, which can then be graphically simulated with several debugging options.

Screen snapshots showing some of the available icons are included in the Appendix.

3.3.1.2 The DE Domain

In the *DE* domain, the events produced by the blocks, which correspond to a change in the system state, are represented by *particle*. The events are processed

by the schedulers in the order of their chronological occurence. Each event has a corresponding time stamp. The *DE* domain is useful for high level system modelling and contains the following sets of stars.

- Sources : They generate signals and can be used to represent external inputs to the system. They include buttons, clocks, and various signal and function generators.
- Sinks : These correspond to system outputs and include text fields, graphs, and interactive displays.
- Control : These stars manipulate interconnections, and include forks, merges and switches.
- Conversion : They include type conversion stars of integer to float and vice versa.
- Queues, servers, Delays : Include delays and stacks.
- Timing stars : These include delay measuring and time-stamping functions.
- Logic stars : As the name indicates, these stars perform logical operations.

In addition, networking and miscellaneous stars are also available. Several menus are available in the graphical editor for design and simulation. The same menu commands can also be typed from the *vem* editor. Details of the commands and the functionality can be found in the Ptolemy users manual [26].

3.4 Summary

In the current chapter, the POLIS hardware-software co-design environment has been described in detail. The main features of the Esterel specification language

and the Ptolemy simulator have been discussed. A brief overview of the system requirements for the software is also presented.

In the next chapter, the stepper-motor controller implementation in software, using POLIS is described. As mentioned earlier, the functionality of the MC33192 hardware chip is replicated in software as far as possible, using the hardware-software co-design methodology of POLIS.

Chapter 4

STEPPER-MOTOR CONTROLLER USING POLIS

4.1 Introduction

As discussed before, the first step in the design process is the system specification. Before we proceed to specify the system in Esterel, we need to understand the functionality of the system. Then we proceed to divide the functionality into suitable modules to be translated to Esterel. Thus, we describe the functionality of the MC33192 and show how we adapt the same to our specification. Then, the rest of the implementation procedure is described. We begin with the operation of a basic stepper-motor, described in the next section, to provide the necessary background.

4.2 A Basic Stepper-Motor

Stepper motors are very popular in computer-controlled systems as they eliminate the need for feeding back positional information. The power source of the stepper needs to be continuously pulsed in specific patterns which determine the speed and direction of a stepper's motion. The motor consists of stator pole pieces and a rotor shaft. The motor operation is achieved by switching the magnetic field of the stator coils causing the magnetic rotor to rotate based on the direction of the magnetic field. Depending on the number of stator coils, the rotation steps of the rotor and consequently the angular frequency can be controlled.

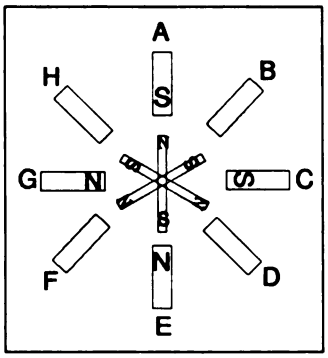


Figure 4.1: Bipolar Stepper Motor

The Figure 4.1 shows a stepper motor which can have fifteen degree increments in motion by suitably wiring the stator pole pieces and switching their polarities. Depending on the number of the segments, very small angular movements of 0.7 to 1.8 degrees are obtained in practical applications of stepper motors. Because of this fixed stepping angle, the position of the motor can be known at any given

time without feedback. This is the main advantage of stepper motors over other dc motors whose position can only be determined by using shaft encoders.

The MC33192 chip is a stepper-motor controller suitable for driving bipolar two-phase motors. In particular, the MC33192 has applications in automotive control systems and is suitable for controlling loads in harsh environments using the MI bus [31]. The functionality of the chip is described next.

4.3 Functionality of the MC33192

For the purpose of this implementation, the objective is to duplicate as many functions as possible from the MC33192. Our specification will then reflect this functionality.

The MC33192[1] [31] is a serial stepper-motor controller which can be controlled by a master micro-controller MC68HC11, hereafter called the MCU, through the MI bus. The MC33192 sub-systems can be broadly classified into two main units, namely the MI bus controller unit and the motor driver unit.

4.3.1 MI Bus controller

This module deals with the push-pull communication sequences with the MCU and acts as the interface between the MI bus and the motor driver. The signal from the MCU consists of five data and three address bits and may be program-

[1]Data sheet attached in Appendix

ming or control instructions. Programming could be address or overwrite bit programming.

Control instructions include full step or half step modes of the motor, and clockwise and counter clockwise direction. Full step is achieved by energizing both coil bridges in the motor driver circuit, while half step is achieved by energizing only one at a time. The motor direction can be changed by reversing the coil current direction. The bus controller functions are briefly listed here.

- Listen on the MI bus
- Convert the serial input signal to parallel
- Follow programming or control instructions from the MCU and send a suitable signal to the motor driver.
- Convert the status signal from the motor driver to serial
- Transmit the status output signal

In addition to this, the bus controller also performs noise and biphase detection of the input signal received from the MCU. So far, these functions have not been duplicated in software.

4.3.1.1 Motor driver

Depending on the instruction received, the motor driver sends a status signal to the MCU through the bus controller and outputs to the two motor coils. The status signal could indicate the following.

- Normal operation
- Programming
- Selection Failed (the MC33192 could not be selected)

In addition, the status signals include two diagnostic signals for thermal and back emf status. This functionality has not yet been incorporated.

4.4 Specification in Esterel

The functionality described in the previous section was specified using the Esterel specification language. Initially, the external interfaces of the entire system were defined. This is the first level of abstraction. Then, the required functionality was divided into suitable modules, and the interfaces of each of the modules were determined, for the second level. This is described here, along with assumptions made whenever required. The system is called SMC, for stepper-motor controller. (Whenever inputs or outputs are indicated without a data type next to them, they are pure signals, which carry no value.)

4.4.1 Level 1 : External Interface

4.4.1.1 Inputs

The MCU sends programming and control instructions to operate the SMC. Programming instructions include address programming, and overwrite bit programming. Control instructions can be direction control, and half step or full step instructions. These instructions are all sent through the MI bus interface. The MI bus mode is determined by bus voltage levels which are also inputs to the

SMC.

4.4.1.2 Outputs

The SMC communicates status codes to the MCU in response to the corresponding MCU instructions. The SMC also provides drive signals to drive motor coils.

4.4.2 Level 2 : System Modules

The system is modularised based on each of the functions that it performs. These may be subsequently combined if found necessary.

The functionality of the SMC considered here, excludes noise and biphase detection. The remaining functions are described below as modules. As before, the interface of each module, i.e., the inputs and outputs of each module are listed and described briefly. Additional signals which are used for debugging purposes only, are listed in the Appendix, but not repeated here.

4.4.2.1 TIMER module

This module generates the timeslots required for the MI bus protocol. Communication on the MI bus is performed by the MCU by sending data in a specific format, with each bit being coded, and sent in a fixed time slot of 25 micro seconds. The timer module generates a signal at the end of each interval.

- Inputs

  Esterel has no physical time attributes. However, we use a millisecond time unit generated by an absolute clock, provided by Ptolemy. This input is fed

into a 68HC11 free-running counter(*frc*), as the ECLK signal. Then, the count signal is received by a 68HC11 output-compare(*oc*) unit, which produces signals whenever required time is elapsed. These modules are available as ready-to-use stars from the Ptolemy palette as 68HC11 peripherals. The *oc* unit receives a start time value from any other module, awaits the time delay and emits an output.

input ECLK, OC3_START(integer) (We use the output compare unit 3 of the 68HC11.)

- Outputs

  The *oc* unit receives a start time value from any other module, computes the time delay and emits an output.

  output OC3_END

4.4.2.2 BUS module

This module manages the communication between the MCU and SMC and continuously watches the time and the bus voltages. For duplicating the serial signals from the MCU, we use a parallel-to-serial converter function.

- Inputs

  Depending on the MODE signal, the SMC is in programming or control mode and accordingly the *bus* module receives status signals from the respective modules. The BUS_SIG input determines the bus level.

  input OC3_END
  input MODE (boolean)
  input PROGRAM_STATUS(integer)
  input CONTROL_STATUS(integer)
  input BUS_SIG(boolean)
  input RESET

- Outputs

  The PULL_FIELD value corresponds to the status signal transmitted to the MCU. The o_BUS_SIG is the bus level transmitted to the MCU. The

PROGRAM_CODE and CONTROL_CODE signals are transmitted to the program and control modules respectively.

output OC3_START(integer)
output PROGRAM_CODE(integer)
output CONTROL_CODE(integer)
output PULL_FIELD(integer)
output oBUS_SIG(boolean)

4.4.2.3 PROGRAM module

This module calls a C function to perform either address or overwrite bit programming. As mentioned before, this module receives the PROGRAM_CODE signal from the *bus* module and returns the PROGRAM_STATUS.

- Inputs input PROGRAM_CODE(integer)
- Outputs output PROGRAM_STATUS(integer)

4.4.2.4 CONTROL module

This module calls a C function to perform control operations based on MCU inputs. Control operations can proceed only when programming is done. As in the case of the program module, the control module receives the CONTROL_CODE and returns the CONTROL_STATUS to the *bus* module. In addition, the control module also generates four motor output signals.

- Inputs
  input CONTROL_CODE

- Outputs

  output CONTROL_STATUS(integer)
  output MOTOR_COIL_A1(integer
  output MOTOR_COIL_A2(integer)
  output MOTOR_COIL_B1(integer
  output MOTOR_COIL_B2(integer)

4.5 Compilation

The complete Esterel file listings for the project are attached in the Appendix. The files were compiled using the strl2shift command and input to POLIS. The shift files are read into POLIS using the *read_shift* command. Then, the *build_sg* and the *sg_to_c* commands build the software graphs (S-graphs) and synthesize the C-code respectively. The *write_pl* command is used to generate the Ptolemy simulation files. These commands can be given in POLIS manually or one can use unix **makefiles** with suitable macros. The makefiles are convenient to use especially for large projects.

4.6 Functional simulation in Ptolemy

The **.pl** simulation files generated previously, are input to the Ptolemy environment. At first, the **pigi** graphical editor is used to enter a graphical netlist of the system, by instantiating individual modules and inter-connecting them. As discussed before, the architectural hierarchy in Ptolemy is named with galaxies

and stars, with a galaxy being the top most level. Thus in this system, we have an **SMC galaxy**, comprising of the **timer, bus, control** and **program** stars in addition to some ready-to-use Ptolemy stars for timing and other utility generation.

The testing of the galaxy is performed using a *target* of Ptolemy, created as a test bed, containing signal sources, and various displays. Since the SMC is designed for communication with an MCU, the schematic includes an *mcu* galaxy, consisting of the following modules. The italicized terms will be used to refer to these modules hereafter, for convenience.

- The **mcu** module *mcu*
- The **OC2** timer (output compare 2) *mcu timer*
- The **bytein** module for parallel to serial conversion *ptos*
- The **OC1** timer (output compare 1)*ptos timer*

The operation of these modules is described in chapter 6, while presenting simulation results and details. These modules comprise the *driver* for the SMC software.

The schematics, or the graphical netlists are included in the Appendix, along with simulation runs. For functional simulation, all the stars were mapped to hardware and tested for an 8 MHz clock. The 68HC11 peripherals used are the free-running counter and the output compare unit 3. These stars are part of the **timer** module. These modules are customized by selecting suitable Ptolemy parameters like the clock pre-scaling factor of the timer counter. The simulation is run in the *debug* mode and provides textual or graphical animation of the run. The following steps describe the functional simulation procedure in detail.

- Create the SMC schematic, by interconnecting the timer, bus, program and control stars. These stars are created using the *make star* command in Ptolemy. The timer stars, viz., *frc* and *oc_prog_rel*, corresponding to the

free-running counter and the output compare unit, are instantiated from the POLIS-PERIPHERALS palette. Include the *mcu* galaxy, which in turn contains the schematic of its component modules mentioned above.

- Create the test_SMC universe, using the SMC galaxy, for the purpose of running the simulation. Suitable sources and sinks are connected to the galaxy in order to run the simulation and display the outputs. These are chosen from the *DE* signals and sinks palette.
- Enter all the parameters for each item, including "HW" implementation, CPU clock, 68HC11 processor and Round_Robin scheduler. These screen snap-shots are also included in the Appendix.
- Verify the functional correctness of the simulation by testing the output for various inputs. In the chapter 6, the test bench and the results of the simulation runs are presented in a tabular form.

4.6.1 Hardware/Software mapping

Ptolemy offers a convenient platform to perform functional simulations without actually implementing a design, by mapping each module to hardware or software. In the SMC design, it could be noticed that once the implementation was chosen as software, the simulation slowed down. Also, the performance of the *timer* module and the *bus* module improved substantially with the hardware implementation. Thus, the simulation process provides a useful aid to the decision-making process of choosing a hardware or software implementation of a given module.

Users can test the simulations for any number of implementations and choose the most suitable one depending on performance and timing requirements. For this project, since the implementation is in software, this step is not explored further, and only test cases are presented in the results.

4.6.2 Software Synthesis and the RTOS

Once the simulation and design partition are satisfactory, the next step is to translate the Ptolemy netlist into a SHIFT netlist. This is done using the *ptl2shift* command. Alternatively, the makefile can be modified suitably to generate this file [4]. This shift file is read into POLIS, and synthesized into software graphs (S-graphs) and finally C code.

An S-graph is a *directed acyclic graph* or *dag* with a set of vertices. It is a processor-independent optimized implementation of the desired behavior and is used for cost estimation in terms of execution time and code size, for a given processor. Each vertex of this graph corresponds to a statement of the synthesized C-code. The cost estimator appends the execution times against each statement of the C-code. [4]

Then the *gen_os* command is used to generate application specific the real-time operating system (RTOS). The RTOS consists of a scheduler and I/O drivers for the design. For this project, as the target micro-controller is the 68HC11 on a handy board, (described in the next chapter) the system setup files and hardware initialization files in the POLIS libraries were customized accordingly. Parameters which need to be modified include the memory map and I/O assignments. The *gen_os* command offers suitable options to use modified versions of the library files.

The steps involved in software synthesis and RTOS generation are described

in the following list.

- Translate the Ptolemy netlist into a SHIFT netlist.
- Read the SHIFT file in polis.
- Set the architecture to 68HC11. (These parameters are easily specified using the makefile.)
- Use the timing and cost estimation commands in POLIS to generate cost estimates for each module on the specified processor. This is useful if the designer needs to choose between various processors.
- The *build_sg* command is used to build the software graphs for each of the modules. These are then converted to C-code using the *sg_to_c* command. The C files are generated in the user specified directory.
- The RTOS is generated using the *gen_os*. Each software CFSM results in a software task. POLIS offers commands to customize the RTOS to implement interrupt routines and configure I/O ports for the target micro-controller. The micro-controller peripherals used in the simulation are implemented as initialization routines called by the RTOS.
- The final step is to make executables for the target micro-controller. For this project, the *Introl* compiler was used with the introl-compatible make files generated by POLIS. The individual C files are compiled and linked with the RTOS to make an executable for the 68HC11.

Since the SMC is entirely mapped into software, the hardware synthesis and prototyping support in POLIS are not described here.

4.7 Summary

This chapter deals with the main focus of this thesis, which is the hardware-software co-design methodology of POLIS as applied to the stepper-motor controller application. To begin with, the specification of the SMC was described in detail, for the overall system and for the individual sub-systems. Then the

Ptolemy simulation procedure, and hardware/software mapping were discussed. The previous section described the software synthesis step which involves many hardware-specific steps, in this case for the 68HC11 micro-controller. The results of this implementation are presented in Chapter 6. In the next chapter, the implementation of the SMC using the MC33192 is described in detail.

Chapter 5

THE MC33192 IMPLEMENTATION

5.1 Introduction

The MC33192 stepper-motor controller generates four phase signals to drive two-phase motors in half or full step mode. (Full-step involves energizing all stator coils, while half-step provides for a smaller angle of rotation by energizing only some stator coils of the stepper motor.) The data sheet of the MC33192 [31] describes the operation of the chip in detail. We discuss the main aspects of functionality here.

The MC33192 belongs to a family of the MI-bus or the Motorola Interconnect bus devices. The MI-bus is normally used in automotive electronics to control loads in a harsh environment. A master MCU can control several devices on a single MI-bus.

5.2 Main Features of the MC33192

The functionality of the MC33192 can be divided into two main units, viz., the **MI Bus Controller** and the **Motor Driver** [31], [32]. The **MI Bus Controller** performs the communication with the MCU, while the **Motor Driver** converts the MCU instruction to the appropriate motor coil signals. The figure 5.1 [32] shows these two units in the MC33192 block diagram. We now take a closer look at the MI bus communication between the MCU and the slave, i.e., MC33192.

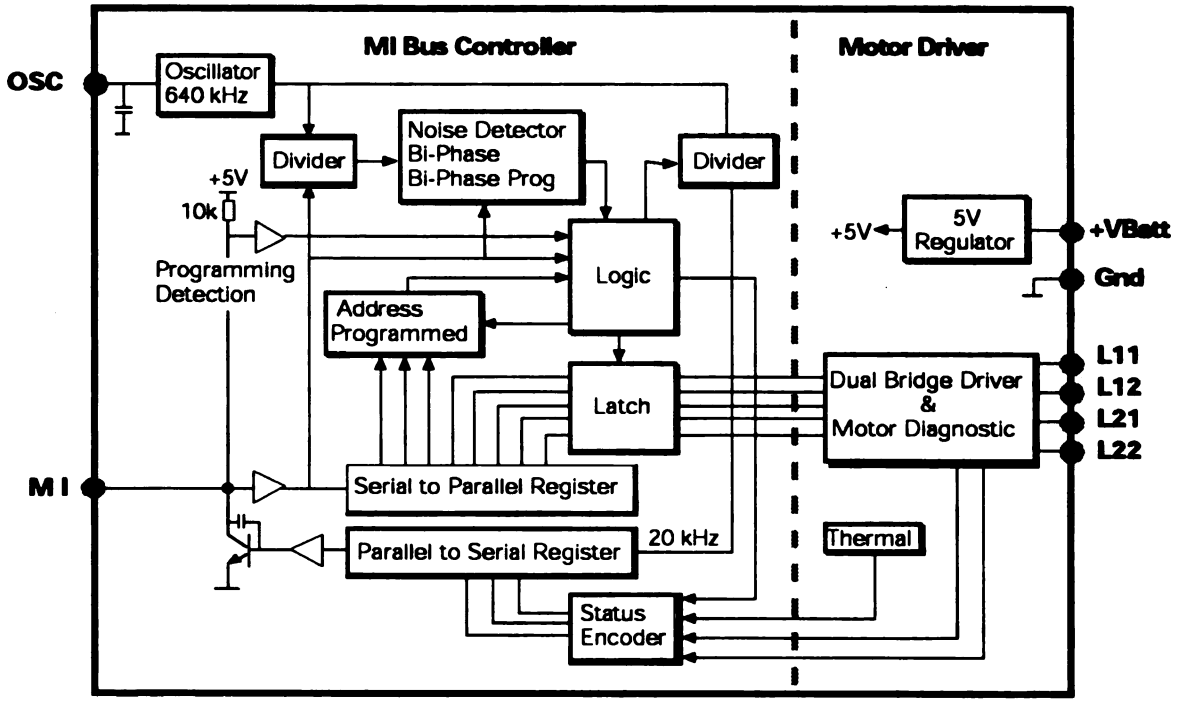


Figure 5.1: Stepper Motor Controller Block Diagram [32]

5.2.1 MI Bus Protocol - Message Format

The MI bus protocol uses a *Push-Pull* technique for message transfer. The *push* sequence is essentially the message sent to the slave device by the master, while the *pull* sequence refers to the message received by the master.

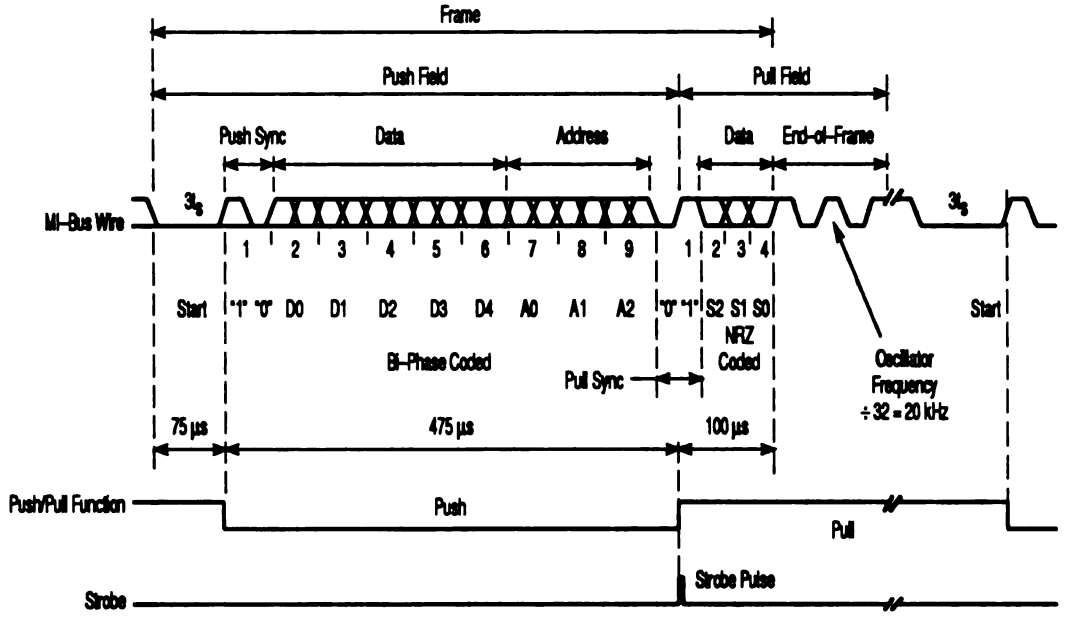


Figure 5.2: Message Coding In The MI Bus Protocol [32]

The messages have a fixed format. Figure 5.2 from [32] shows these sequences. The components of the message are as follows.

- *Start bit* The MCU takes control of the bus by issuing a start bit, which holds the MI bus at a logical zero state for three consecutive time slots.
- *Push Field* The push field contains the MCU message with a push-sync bit, five data bits and three address bits, followed by a pull-sync bit, as shown in the figure 5.2.
- *Pull Field* The pull field contains the serial data read by the MCU from the slave device. It contains three status bits and an end-of-frame signal.

5.2.2 Message Coding

The push field bits are coded using the Manchester bi-phase code. The bi-phase code uses two time-slots to encode a single bit.

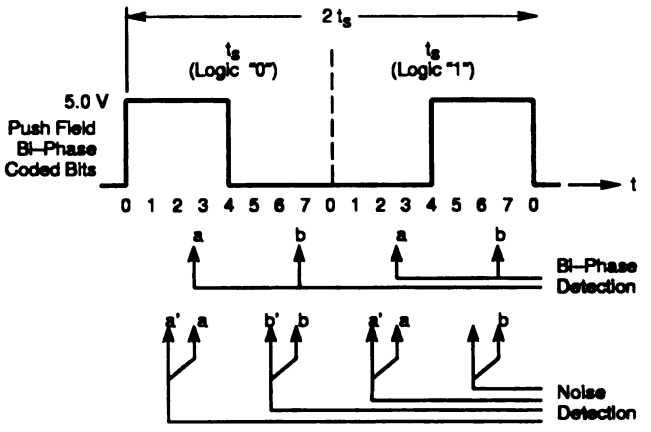


Figure 5.3: Bi-Phase Code [32]

Figure 5.3 shows the logic levels "1" and "0" represented using bi-phase code. This enables bit-wise error detection by the slave device which uses an exclusive-OR detector circuit for the push sequence.

The pull field uses a non-return to zero or NRZ code. This encoding uses a high value represents logic "1" and a low value represents logic "0".

5.2.3 Address Programming

Each MC33192 on the MI bus is programmed with a specific address by the MCU. The address programming sequence is performed in three steps.

5.2.3.1 Step 1

- The MI bus is supplied at 12 volts.
- The MCU pushes the address to be programmed, with the data bits set to 0.
- The MCU checks the status bits for the **programming code 110**.

5.2.3.2 Step 2

- The MI bus remains at 12 volts.
- The MCU repeats the instructions in step 1.

5.2.3.3 Step 3

- The MI bus is supplied at 5 volts.
- The MCU pushes the address to be programmed, with the data bits set to 0.
- The MCU checks the status bits for the **OK code 100**.

Steps 1 and 2 are repeated until the 110 code is received. All three steps are repeated until the 100 code is received.

5.2.4 Overwrite-bit Programming

This is performed in two steps.

5.2.4.1 Step 1

- The MI bus is supplied at 12 volts.
- The MCU pushes the address to be programmed, with the data bits set to 0 except D4, which is set to 1.
- The MCU checks the status bits for the **programming code 110**.

5.2.4.2 Step 2

- The MI bus is supplied at 12 volts.
- The MCU repeats the instruction in step 1.
- The MCU checks the status bits for the **OK code 100**.

Steps 1 and 2 are repeated until the **OK code 100** is received. Programming failure occurs if the necessary code is not received after 8 repeats.

5.2.5 Motor Control

The Motor driver unit of the MC33192 consists of two H-bridges to provide the motor drive signals. The H-bridge circuits are described in the device data sheet attached in the Appendix [31]. Essentially, the when only one of the bridges is active, the motor is in half-step mode, and when both are active, the mode is full-step. The direction of the motor is controlled by the sequence of the current direction in the H-bridges.

Once a device is programmed, the MCU issues control instructions. The five data bits in the push field are used to determine motor direction and the step mode.

5.3 Implementation of the MC33192

5.3.1 Hardware Setup

The figure 5.4 shows the hardware setup of the implementation [32]. The MCU board used here is a handy board [33] with a 68HC11. The figure also shows the MI bus interface, which consists of a single npn-transistor.

Figure 5.5 [33] shows a block diagram of the handy board, while figure 5.6 [33] shows a schematic of the handy board.

The user interface consists of the following:

- Inputs The *program_enable* is a user input to begin programming. The *direction*, *address* and *step* are user inputs to control the motor direction, address to be programmed, and half or full step size respectively.
- Outputs The *program status* bits indicate the current status of the program. The meaning of the status bits is defined as per table 5.1 [31] below.

Table 5.1: Program Status Bits

S2 S1 S0	Status
000	Not used
001	Enable programming
010	No back emf
011	Not used
100	Normal
101	Thermal
110	Programming
111	Error

The signal from PD2 of the handy board is used to set the MI bus at 12 volts for the programming sequence.

5.3.2 MC33192 Software

As mentioned before, the software to drive the MC33192 was developed on the MC68HC11 microcontroller for this project. The software was developed in assembly language[1] and downloaded on the handy board using the Interactive-C [34] binary format. The next few paragraphs describe the design of the software program. A complete listing of the program is attached in the Appendix.

5.3.3 Software design

The software program is divided into three modules, viz.,

- Initialization module
- Timer Interrupt module
- Main module

These individual modules are now described.

5.3.3.1 Intialization

This module performs the following tasks.

- Initialize I/O ports, reset timer and output compare unit
- Initialize variables and stepper motor parameters
- Initialize timer interrupt routine
- Set controller mode to program mode or control mode
- Set initial push-sequence depending on control or programming mode and user-inputs.

[1]The assembly code is written specifically for use with Interactive -C.

5.3.3.2 Timer Interrupt

The timer interrupt module is used to generate an interrupt request every 5 msec, and execute the appropriate timer interrupt service routine. For this purpose, the output compare function of the 68HC11 is used. Whenever the free-running counter of the MCU matches the value in the output compare register, an interrupt request is generated and it is serviced by an interrupt service routine.

5.3.3.3 Main

The main execution module consists of the core of the software program. Each time the timer interrupt is generated, control is transferred to an appropriate interrupt service sequence in the main module.

In the interrupt service routine the MCU issues the required instruction, reads the status information and prepares the next instruction. Then, the address of the interrupt service routine to execute the next instruction is updated. Then the program control passes once again to the timer interrupt module to generate the next interrupt.

The figure 5.7 describes the program operation in the form of a flow chart. The key steps in the main module, which are the transmission of the push-sequence, and the analysis of the pull-sequence, are described further here.

5.3.3.3.1 Push Sequence As shown in the figure 5.7, once the initialization of the program is completed and the mode of the controller is set, the control of

the program passes to the main module at the next timer interrupt.

The first step in the main module, is to convert the push sequence to bi-phase code, and transmit it to the slave in a fixed time. Each message time slot used for this project is 25 micro seconds, as recommended in the MC33192 data sheet. At the end of the push sequence, the MCU listens for the pull field sent by the slave device and this is analyzed as follows.

5.3.3.3.2 Pull Analysis The purpose of the pull analysis is to determine the next push sequence and display the status information to the user. The pull-field status value returned by the slave device corresponds to one of the following two cases:

Case I The status corresponds to programming or normal operation. In this case, the push-sequence is updated and the status if displayed to the user. The interrupt service routine is updated to perform the next step in the motor control or programming, at the next timer interrupt.

Case II The status corresponds to any of the abnormal conditions listed in table 5.1. In this case, the push-sequence is not updated, the corresponding error code is displayed to the user, and the control is transferred to the timer interrupt software directly. The next timer interrupt does not cause any data transmission to the slave device. Normal operation continues only after the error is fixed and the MCU is reset.

5.4 Summary

This chapter deals with the stepper-motor controller implementation using the MC33192. To begin with, the main features of the MC33192 operation and the MI bus protocol were explained. Then, a description of the hardware setup with the necessary schematics was presented. We then described the design and implementation of the software program to control the MC33192. This chapter concludes the description of the two approaches to implement the stepper-motor controller used in this thesis. These are the software implementation using POLIS-Esterel-Ptolemy environment, and the hardware version using the MC33192. In the next chapter, the results of these two implementations are presented, and analyzed.

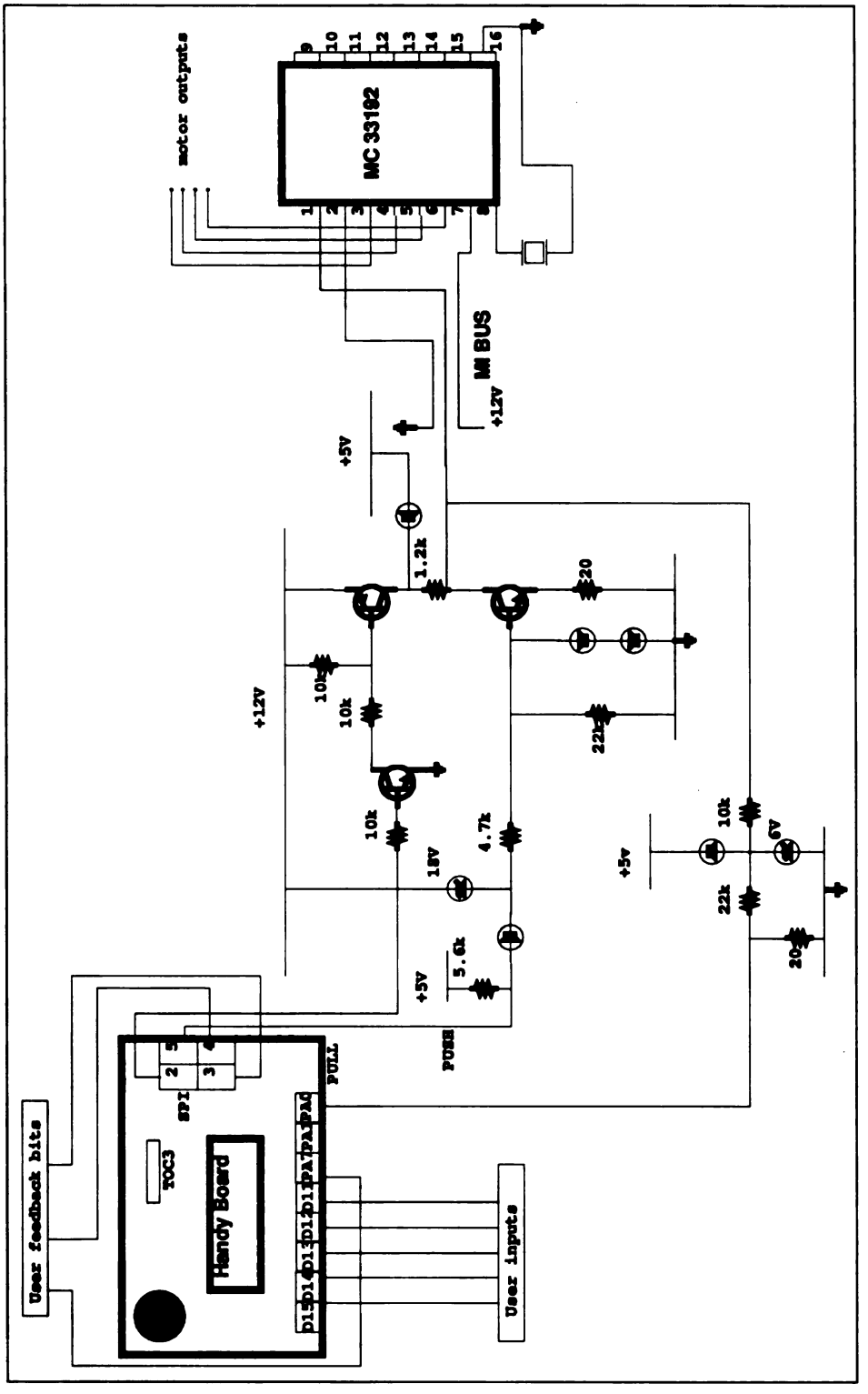
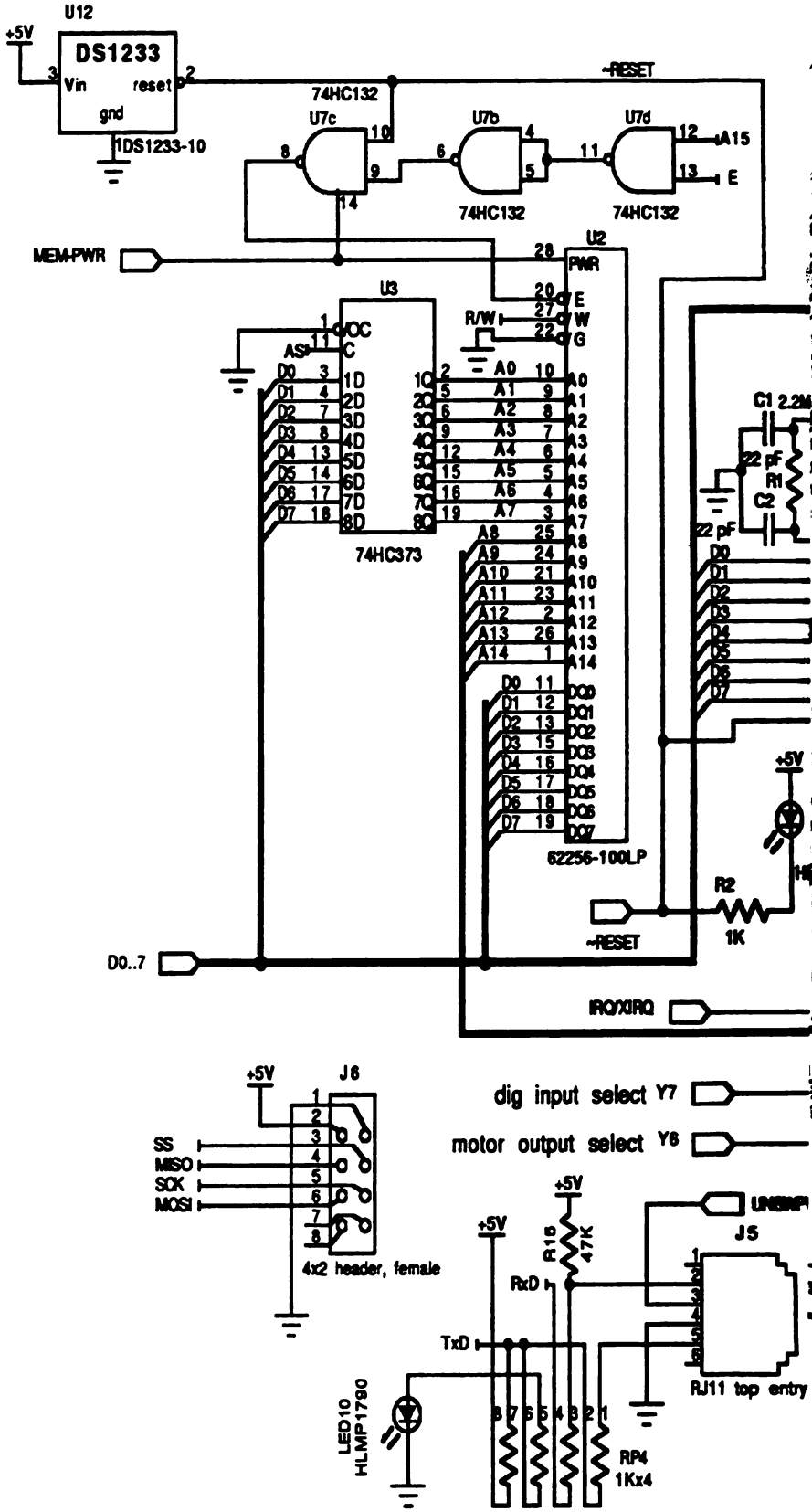


Figure 5.4: Hardware Setup For The MC33192

Figure 5.5: Block diagram Of The Handy Board [33]

Figure 5.6: Schematic Of CPU And Memory [33]


U12
+5V
DS1233
Vin
reset
gnd
DS1233-10
~RESET
74HC132
U7c
U7b
U7d
74HC132
74HC132
A15
E
MEM-PWR
U2
PWR
U3
R/W
74HC373
62256-100LP
C1 2.2M
22 pF
R1
C2
22 pF
R2
1K
~RESET
D0..7
IRQ/XIRQ
+5V
J6
SS
MISO
SCK
MOSI
4x2 header, female
dig input select Y7
motor output select Y6
+5V
R15
47K
RxD
TxD
J5
RJ11 top entry
LED10
HLMP1790
RP4
1Kx4

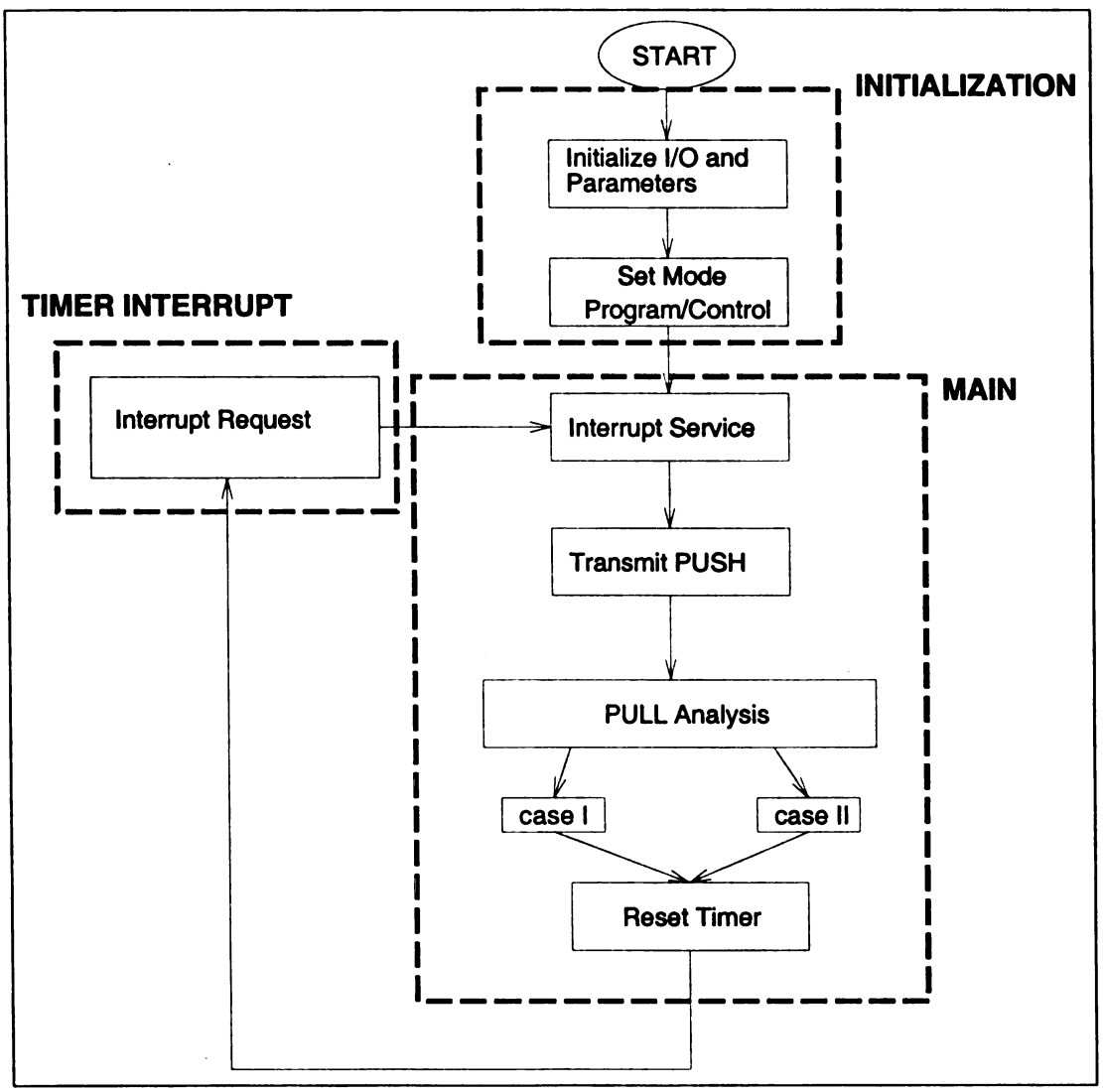


Figure 5.7: The Software Program

Chapter 6

RESULTS, ANALYSIS AND CONCLUSIONS

6.1 Introduction

In the previous two chapters, two different implementations of a stepper-motor controller have been described. One of them used an existing hardware chip, the MC33192, and the other used a hardware-software co-design approach. In this chapter the results of the two implementations are presented and they are analyzed with respect to some of the issues involved in hardware software co-design. First, we discuss the results of the MC33192 version, and then the POLIS implementation.

6.2 The MC33192 implementation

The performance of the MC33192 software was verified for specific inputs and outputs as per the specification. Table 6.1 indicates these inputs and outputs. The motor modes are achieved as follows:

- step=0/1 indicates half/full step.
- dir=0/1 indicates clockwise/counterclockwise (cw/ccw)
- addr=0/1 indicates the chip address is 2 or 4. (arbitrary)

The step sequences shown in the table are for cw operation. They are reversed for ccw operation.

Table 6.1: MC33192 Testing

MOTOR mode	Program status[1]	Coil outputs[2]
Not running	Programming	None
Half step	OK	H H H Z H L Z L L L L Z L H Z H
Full step	OK	H H H L L L L H

In the next section, we discuss the POLIS implementation.

[1]The status codes are listed in table 5.1

[2]H, L, Z indicate forward, reverse and high impedance states of the coils

6.3 The POLIS implementation

As before, this implementation will be referred to as the SMC(stepper-motor controller). The results of the SMC are presented in two sections - one the Ptolemy simulation results for various test conditions; and the other the test results using the 68HC11.

6.3.1 Ptolemy simulations

In order to test the Ptolemy simulation, a set of important criteria were laid down. These were necessary to ensure that all the critical aspects of the design were tested, before proceeding with implementation. This is in keeping with one of the goals of this thesis, i.e., to explore the POLIS hardware software co-design methodology.

As described in chapter 3, Ptolemy offers two useful tools, in addition to other displays, to trace simulation runs. These are the *firing file* and the *overflow file.* The location of these files can be specified as parameters for the test bed universe. These files provide two key sets of information for the simulation. The *firing file* indicates the time stamp of firing of all software stars. The *overflow file* on the other hand, indicates all the missed events for each module, which may occur due to asynchronous operation of the system modules. With this background, the following objectives were to be achieved through the tests.

- Verify functional correctness.

- Timing analysis by varying clock frequency and communication time-slots.
- Explore various hardware software partitions. Compare an all software simulation to a combined hardware/software partition.

We now discuss each of these in detail. The second and third items in the above list of criteria have been combined, as timing behavior is also dependent on whether a particular module is implemented in hardware or software.

6.3.1.1 Functional Verification

The test bench for functional verification was based on the operation of the SMC itself. This constituted testing the outputs corresponding to each input in the control or programming modes. The control panel schematic indicating this interface for the test bed is attached in the Appendix. In all cases, the operation was as expected. Table 6.2 indicates the results obtained for the simulation. For the functional simulation, all the modules were mapped to hardware. The *overflow* file was verified to ensure that no events were missed.

6.3.1.2 Timing Analysis and HW/SW partitions

For this thesis, the final implementation of the SMC is software running on the 68HC11. Thus the clock-frequency of the target micro-controller was already known to be 8 Mhz(crystal frequency), with a system clock (ECLK) of 2 Mhz. However, in order to emulate a real world design, where architecture decisions have

[1] cw: clockwise, ccw: counter clockwise, H, L, Z indicate forward, reverse and high impedance states of the coils

Table 6.2: Functional Simulation Using Ptolemy

Mode	Motor status[1]	Coil outputs A, B	Comment
1 (Programming)	Not running	None	Pull value=6; Addr value = 2 or 4
0 (Control)	Half step	H H H Z H L Z L L L L Z L H Z H	step=0;The sequence is reversed for ccw.
0 (Control)	Full step	H H H L L L L H	step=1;The sequence is reversed for ccw.

to be made by designers, the SMC behavior was analyzed for alternative clocks, and with alternative partitions. Table 6.3 indicates the results obtained for some partitions and clocks. This is followed by a brief explanation of the results. The primary goal of this exercise is to gain insight into the uses of the Ptolemy simulator and the criteria for design optimization. These issues are discussed towards the end of this chapter in the analysis and conclusions sections.

Table 6.3: Clocked Simulation Using Ptolemy

Partition	CPU clock	Time slots	Step cycle
All HW	500ns	200/25μ s	$\geq$ 250μ s
All SW[1]	5μ s	200/25μ s	4.5ms
	2μ s	250/25μ s	3.6ms
	1μ s	250/25μ s	5.1ms
	500ns	275/25μ s	10ms
All SW except *mcu, bus*	500ns	50/25 μ s	5ms

Table 6.3 presents the results of the time-based simulation of the SMC system. The *partition* indicates which modules are mapped to hardware, and which to software. The *CPU clock* corresponds to the ECLK signal. The two *time slot* values correspond to the push transmission time slot and the pull reception time slot. The *step cycle* corresponds to the time taken for transmitting a push sequence and receiving status bits. All the values in the tables only specify only upper or lower limits of the timing parameters. Any values which are suitable multiples of the existing values would also provide similar results. The next section presents a detailed analysis of the above results.

The all-software partition is used to generate the downloadable file for the 68hc11 using the POLIS makefiles, as described in chapter 4. In the next section a detailed analysis of each of these results is presented.

[1]The 68hc11 peripherals(timers) are mapped to behavioral mode

6.4 Analysis

We now examine the co-design approach used for this thesis, in the light of the results obtained for each implementation of the stepper-motor controller. In particular, the focus is on co-simulation and hardware/software partitioning.

6.4.1 Background

It is in order here, to note some important characteristics of the *DE domain* in the Ptolemy simulation [26].

- *Time* refers to simulated time.
- The simulation is *event-driven* with each event being time-stamped and queued in a chronological order.
- The *DE* scheduler processes events in the queue at run-time and *fires* or executes the appropriate star.

As described earlier, the "firingfile" indicates the time stamps of the execution of software CFSMs. An excerpt of the firingfile looks is included next.

test_hbSMC.hbSMC1.hbMCU1.hbmcu1: 0 0 start

test_hbSMC.hbSMC1.hbMCU1.hbmcu1: 290 -1 end

test_hbSMC.hbSMC1.hbbus1: 7616 0 start

test_hbSMC.hbSMC1.hbbus1: 7771 -1 end

test_hbSMC.hbSMC1.hbMCU1.hbmcu1: 7771 0 start

This shows the name of the star, and the time stamp for the execution. If a star takes too long to respond to an event, the event may be over-written by the

time the star is ready. These missed events or "overflows" can be tracked down using the overflow file, which indicates which events are missed. Then the timing parameters of the module in question can be suitably altered in order to achieve accurate responses. It is necessary to mention however, that some missed events could be insignificant, while others may be critical. An excerpt of the overflowfile is included next.

test_hbSMC.hbSMC1.hbMCU1.hbbytein1: 17690 e_clock

test_hbSMC.hbSMC1.hbMCU1.hbbytein1: 19130 e_clock

test_hbSMC.hbSMC1.hbMCU1.hbbytein1: 20090 e_clock

Here, the star name, the time stamp of the event and the event name are indicated.

Also, as mentioned before, the POLIS system exhibits *asynchronous* execution of concurrent modules. Thus, if synchronization is required between such modules, this can be achieved using explicit handshake mechanisms or other suitable interfaces.

6.4.2 Co-simulation

In a system with hardware/software modules, the co-simulation environment provides a useful tool to verify the system behavior and optimize design criteria, without actually building the system. Some of these are

- Choice of HW/SW implementation for any given system component.
  This could be an architecture or partitioning issue. The given system may or may not have the flexibility for major changes at the simulation stage.

However, various HW/SW alternatives can be explored and the system performance can be accordingly evaluated.

- Synchronization between modules

  This hinges on two important issues. First, the timing requirements specified for the system. If no trade-offs are possible, then the strict synchronization requirements must be met in order to ensure required performance. This may mean sacrificing cost or flexibility.

  Second, the interfaces designed in the system. At the design stage, some parameters or some components may be considered more critical than others for optimal system performance. However, it is only during the co-simulation stage that the behavior of the system becomes clear. It is here, that the behavior of individual modules or sub-systems containing several modules, can be isolated to analyse their contribution to overall system performance. This enhances the quality of the system design and increases confidence in the final product.

- Performance vs Cost trade-offs.

  Trade-off considerations arise at every stage of system design. At the specification stage, trade-offs could be about the requirements themselves. These could be influenced by available resources and criticality of the system in consideration.

  At the architecture stage, these decisions have a direct bearing on the final system cost. They also determine the partitioning decisions which follow the simulation step.

In the following sections, some of the above criteria are explored further for the stepper-motor controller case.

6.4.3 The SMC system

It is instructive to re-organize the stepper-motor controller system solely based on the nature of the tasks performed by each component of the system.

6.4.3.1 Timing behavior

The *mcu* sub-system and the *bus* sub-system perform operations involving sequencing, and are heavily dependent on timing performance. The program and control modules are compute-intensive and have comparatively lighter loads. Accordingly, once these two modules were cleared during the functional simulation, they do not significantly change their behavior when they are mapped to software. This was demonstrated during the simulations and these modules performed as expected.

In the *mcu* and the *bus* sub-systems, the modules in question are the *mcu* and the *bus* modules and the timing modules.

The timing performance of these sub-systems involved two different timing criteria. One was the response of an individual module with changing timing parameters. The other was the relative timing or synchronization between different modules.

We illustrate each of these here. As the clock frequency was changed over 5μs to 500ns , more and more missed events occurred in the *mcu* subsystem with the given timer setting. This was reflected in the overflow file. For a faster clock, the timer settings needed to be increased in order to avoid missed events. This problem occurs for those events where there is no explicit acknowledge.

As mentioned before, this is due to the asynchronous behavior in POLIS. The solution is to have explicit handshake mechanisms in the code, or to space

important timer events sufficiently, so that clock is not too fast for the response. In this case, we use the latter method, and increase timer periods sufficiently to eliminate or reduce missed events. The preceding paragraphs deal with the *mcu* module alone.

Now, we move on to the more complex inter-module timing dependencies. This involves the *mcu* module, parallel to serial or *ptos* module and the *bus* module.

- If the parallel to serial converter module reads data too fast or too slow, the result would be that same *mcu* data is re-read, or some data values may be missed.
- Further, if the *bus* module is scheduled incorrectly it may read junk values from the ptos converter, thus causing incorrect or zero outputs at the drive signals.

A closer analysis of the above leads us to the conclusion that of these two interdependencies, the latter is more crucial because a missed step for the motor is more tolerable than an invalid or absent output signal. Accordingly, the timers were adjusted to eliminate invalid data first, and then to eliminate missed events.

First the *ptos* timer was reset to a suitable value to eliminate erroneous reads. This value is indicated by first time slot in the table 6.2. This could mean that some values are missed, but all the output data is valid.

Then, the *mcu* timer period was increased with faster clocks, with an approximate estimate. This value is denoted by "step-cycle" in table 6.2. This helped eliminate the missed values also, though at the cost of lower speeds. This procedure worked most of the time for all the cases. However, in some cases both

timers needed adjustment. The *ptos* timer setting was also revised to achieve correct performance. In any case, the strategy proved effective in arriving at correct functionality for the various frequencies tested.

It was previously observed there is no standard tool for hardware software codesign. Accordingly some design aspects are very specific to the codesign environment used. This can be illustrated by the following, with respect to this project.

If timing criteria are well known at the outset, timing interdepencies could be reduced during the specification stage itself. Thus, in our case, the *mcu* module and the *ptos* module could be combined, in order to eliminate their interface. However, this defeats the purpose of an "implementation-independent" specification. On the other hand, it could improve the system performance. Again, some modularity would be sacrificed. This is evident in the *bus* module, which has a serial to parallel converter built-in, and allows 25μ s resolution.

As mentioned before, all critical interface signals could be modified to have a built-in hand-shake mechanism to reduce missed events. This could significantly increase code size, and result in over-specifying the system. Thus, all these decisions involve several system-dependent factors.

6.4.3.2 Hardware or Software

As indicated before, timing analysis and hardware/software partitioning are related due to better performance of hardware under strict timing requirements. This factor is dealt with in the next section. Here, we examine other distinguishing features of hardware and software particularly in the co-design case.

- **Flexibility**

  Since software can be altered, while hardware needs to be replaced, in general we can consider software to be more flexible than hardware. An example for the SMC system could be that the communication time slots could be increased, in order to simplify the code, while achieving desired performance. However in the MC33192, the MCU software needs to be tailor-made for the required time slots. In this sense, software implementation obviously provides some degree of "*programmability*" which hardware does not.

  However, in an application-specific embedded system, this could be a more complex issue. For instance, the software may be specially written for a piece of hardware, and rewriting the software may need significant changes in the hardware/software interfaces.

- **Reusability**

  Code re-use is an important feature of system design. It may provide significant savings in large systems. In case of the SMC, the system could be extended for multiple motors, or multiple outputs or extended functionality by re-using existing code. This could be at the system level or modular level. Using co-design, the new system could be optimized for various parameters before the final implementation. In case of hardware, the only option is additional hardware and the system may not be upgradable easily.

  Some practical considerations in replacing the MC33192 chip with the SMC could be the following. The SMC may not match the speed of the hardware chip, while being sufficient for the required functionality. In an actual automobile control system, it might be more desirable to separate the MCU from the motor due to the harsh environment. On the other hand, this could also be achieved by providing some additional driver logic at the motor end. Thus the decision could depend on the actual use of the system.

In the preceding paragraphs, several aspects of the timing and performance of the system were discussed in the hardware/software co-design context. The next section presents conclusions obtained from this analysis.

6.5 Conclusions

Many features of the hardware/software co-design were presented in this thesis over several chapters. The work in this thesis serves to demonstrate using the SMC system, some of these characteristics. These include, criteria for co-design, specification, architecture and co-simulation issues.

From these analyses, certain trade-offs in using co-design are evident. The first of these, is the fact that the co-design process is not only application-specific, but to a large extent, environment- specific. This was described in the previous section. It follows, that the hardware-software co-design process, although advantageous, may lead to certain problems. These could be the need for special resources, and time delays for learning curves in new environments.

However, these very reasons make the co-design methodology very powerful, and provide system designers with a number of choices and tools for optimum design which is faster and cheaper.

Chapter 7

RECOMMENDATIONS FOR FUTURE WORK

The features of the POLIS co-design system used for this project provide a reasonable scope and focus on certain aspects of co-design. The next step would be to continue the SMC implementation, and generate hardware/software partitions. These could then be used to generate .xnf files for prototyping on Xilinx boards [4] to fully explore the capability of the POLIS environment. In addition, this would provide the experience to develop the POLIS, Esterel, Ptolemy system into a full-fledged and integrated co-design environment with all the associated tools in one system, for future use in MSU.

APPENDIX

Program Listings

```
module hbbus:
constant BUS_V_TIME: integer, TIME_SLOT: integer;
input OC3_END, MODE(boolean), PUSH_FIELD(integer),
PROGRAM_STATUS(integer),
CONTROL_STATUS(integer),RESET, BUS_SIG(boolean);
output OC3_START(integer), PROGRAM_CODE(integer), CONTROL_CODE
(integer), PULL_FIELD(integer),
PUSH_DEBUG(integer),PUSH_SYNC,PULL_SYNC, oBUS_SIG(boolean);
function Rx(integer,integer) :integer;
function Tx(integer,integer) :integer;
constant count8:integer, bitcount:integer;
signal Receive_push in
var Tcount:=0 :integer,
Bcount:=1:integer,
Curr_push :=0:integer,
Prev_push :=0:integer,
Push_bit:=0 :integer,
Pull_in:=0:integer,
Pull_out:=0:integer,
pcount:=1:integer in
loop
weak abort
await BUS_SIG do
if Bcount6 then
Bcount:=Bcount+1;
if(?BUS_SIG = false) and Bcount¡5 then
Tcount:=Tcount+1;
elsif (?BUS_SIG=true) and Bcount¡5 then
Tcount:=0;
end if;
if (?BUS_SIG=true) and Bcount=5 then
if Tcount=3 then
emit PUSH_SYNC;
end if;
end if;
elsif Bcount¡14 and Bcount¿5 then
Bcount:=Bcount+1;
if (?BUS_SIG=true) then
Push_bit:=1
else
Push_bit:=0;
end if;
```

```
Curr_push := Rx(Push_bit,Prev_push);
Prev_push := Curr_push;
if Bcount=14 then
emit PUSH_DEBUG(Curr_push);
if (?MODE = true) then
emit PROGRAM_CODE(Curr_push);
else
emit CONTROL_CODE(Curr_push);
end if;
end if;
elsif Bcount=14 then
Bcount:=Bcount+1;
pause;
elsif Bcount=15 then
if(?BUS_SIG=true) then
emit PULL_SYNC;
end if;
Bcount:=1;
Prev_push:=0;
Tcount:=0;
if(?MODE=true) then
Pull_in := (?PROGRAM_STATUS);
else
Pull_in := (?CONTROL_STATUS);
end if;
emit OC3_START(TIME_SLOT);
await OC3_END;
emit PULL_FIELD(Pull_in);
pcount:=1;
repeat 5 times
pause;
if pcount¡4 then
Pull_out := Tx(Pull_in, pcount);
if(Pull_out = 1) then
emit oBUS_SIG(true);
else
emit oBUS_SIG(false);
end if;
elsif pcount=4 then
emit oBUS_SIG(true);
else
emit oBUS_SIG(false);
```

```
end if;
pcount:=pcount+1;
end repeat;
end if;
end await;
when RESET
end loop
end var
end signal
.
```

```
module hbprogram:
input PROGRAM_CODE:integer;
output PROGRAM_STATUS:integer,PFUNC_DEBUG : integer;
function PROG_FUNC(integer) : integer;
loop
await immediate PROGRAM_CODE do
pause;
var PROG_IN :=0: integer,
PROG_OUT:=0 : integer in
PROG_IN:= ?PROGRAM_CODE;
PROG_OUT:=PROG_FUNC(PROG_IN);
pause;
emit PROGRAM_STATUS(PROG_OUT);
emit PFUNC_DEBUG(PROG_OUT);
end var;
end await;
end loop
.
module hbcontrol:
input CONTROL_CODE :integer;
output CONTROL_STATUS:integer,
MOTOR_COIL_A1:integer,MOTOR_COIL_A2:integer,
MOTOR_COIL_B1:integer,MOTOR_COIL_B2:integer, A_DEBUG :integer,
B_DEBUG:integer, CFUNC_DEBUG: integer;
function CONTROL_FUNC(integer) : integer, MOTOR_FUNC_A(integer) :
integer, MOTOR_FUNC_B(integer) : integer, PROG_FUNC(integer):integer;
loop
await immediate CONTROL_CODE do
pause;
var Code_in :=0:integer,
Code_out :=0:integer,
Coil_A :=0:integer,
Coil_B :=0 :integer in
Code_in := ?CONTROL_CODE;
Code_out:= CONTROL_FUNC(Code_in);
Coil_A := MOTOR_FUNC_A(Code_in);
Coil_B := MOTOR_FUNC_B(Code_in);
if (Coil_A=1) then
emit MOTOR_COIL_A1(1);
emit MOTOR_COIL_A2(0);
elsif(Coil_A=2) then
emit MOTOR_COIL_A1(0);
```

```
emit MOTOR_COIL_A2(1);
else
emit MOTOR_COIL_A1(0);
emit MOTOR_COIL_A2(0);
end if;
if (Coil_B=1) then emit MOTOR_COIL_B1(1);
emit MOTOR_COIL_B2(0);
elsif(Coil_B=2) then
emit MOTOR_COIL_B1(0);
emit MOTOR_COIL_B2(1);
else
emit MOTOR_COIL_B1(0);
emit MOTOR_COIL_B2(0);
end if;
emit A_DEBUG (Coil_A);
emit B_DEBUG (Coil_B);
emit CONTROL_STATUS(Code_out);
emit CFUNC_DEBUG(Code_out);
end var;
end await;
end loop
.
```

```
module hbmcu:
input DIR:boolean, STEP:boolean, ADDR:boolean, OC2_END, MODE:boolean,
RESET;
output PUSH_VAL:integer;
constant bitcount:integer;
function Get_push(integer, integer):integer;
function Next_step(integer, integer):integer;
loop
var push_seq:=0:integer,
push_val:=0: integer,
addr:=0: integer,
step_count:=0 : integer,
prog_done:=0:integer in
weak abort
pause;
every immediate MODE do
if ?ADDR=true then
addr:=4;
elsif ?ADDR=false then
addr:=2;
end if;
if ?DIR=true and ?STEP=true then
push_seq:=1;
%fcw step_count:=7;
elsif ?DIR=true and ?STEP=false then
push_seq:=2;
%hcw step_count:=8;
elsif ?DIR=false and ?STEP=true then
push_seq:=3;
%fccw step_count:=1;
elsif ?DIR=false and ?STEP=false then
push_seq:=4;
% hccw step_count:=1;
end if;
if ?MODE=true and prog_done=0 then %programming
push_seq:=5;
step_count:=1;
positive repeat 5 times
pause;
await OC2_END do
if (step_count¡4) then
push_val:=0;
```

```
step_count:=step_count+1;
elsif (step_count=4) then
push_val:=8;
step_count:=step_count+1;
elsif(step_count=5) then
push_val:=8;
step_count:=8;
%terminate programming
prog_done:=1;
end if;
push_val:=push_val+addr;
emit PUSH_VAL(push_val);
end await;
end repeat;
elsif ?MODE=false and prog_done=1 then %control
loop weak abort
pause;
await immediate OC2_END;
push_val:=Get_push(push_seq,step_count);
step_count:=Next_step(push_seq,step_count);
push_val:=push_val+addr;
emit PUSH_VAL(push_val);
when MODE;
end loop;
end if;
end every;
when RESET;
end var;
end loop;
.
module hbbytein:
input Push_field:integer, start,clock;
output out_bit:boolean;
constant bitcount:integer;
function Tx(integer, integer):integer;
loop await immediate Push_field;
var PUSH_IN :=0:integer,
PUSH_OUT:=0:integer,
count : integer in
count :=0;
PUSH_IN := ?Push_field;
positive repeat bitcount times
```

```
await clock;
count := count+1;
if count¡4 or count=5 or count=14 then
emit out_bit(false);
elsif count=4 or count=15 then
emit out_bit(true);
else PUSH_OUT := Tx(PUSH_IN, (count-5));
if PUSH_OUT=1 then
emit out_bit(true);
else
emit out_bit(false);
end if;
end if;
end repeat;
end var;
end loop;
.
```

```
/* Programming:
input 0 = 0 data + address say 101 then input byte is 5
output
110 = 6
input repeat as above
output
110 = 6
input repeat as above
output
100 = 4
*/
/* Control:
See page 6 of MC datasheet. for the 8 combinations,
we shift the push three places right to eliminate address
and we list the values of just the data bits, with the order D0,to D4.
The status output for normal operation
is 100 = 4
The coil outputs are
(Here we assume that coil A high means energise A1-A2 and low means A2-A1)
Z is high impedance
INPUT coil A coil B
21 1 1
20 1 2
23 1 0
7 2 0
31 0 0
28 0 2
29 0 1
5 2 1
/* This is either ovrbit programming or address programming.
as per present options, address could be 100 or 010 (A0,A1,A2).
so, pin could be 4 or 2 any other is error.
For ovrbit, the value is 12 for address 4 and 10 for address 2.*/
#define pos 1;
#define neg 2;
#define Z 3;
#define error 4;
static int prog_flag=0;
static int address=0;
int PROG_FUNC(int pin){
int pout=0;
if (prog_flag ==0){
```

```
if(pin==4——pin==2){
address=pin;
pout = 6;
prog_flag=1;
/* 6 */
} }
else if(prog_flag==1){
if(pin==address){
pout=6;
/* 6 */
prog_flag=2;
}
}
else if(prog_flag==2){
if(pin==address){
pout =4;
/* 4 */
prog_flag=3;
}
}
else if(prog_flag==3){
if(pin==(address+8)){
pout=6;
/* 6 */
prog_flag=4;
}
}
else if(prog_flag==4){
if(pin==(address+8)){
pout=4;
/* 4 */
prog_flag=5;
}
}
else{
prog_flag=0;
pout=7;
return 7;
}
return pout;
}
int CONTROL_FUNC(int cfin){
```

```
int cfout=0, cfdata=cfin;
int mask_addr=7;
mask_addr&=cfin;
if(mask_addr==4||mask_addr==2){
cfdata ≫=3;
if (cfdata==5||cfdata==7||cfdata==20||cfdata==21){
cfout = 4;
} else if(cfdata==23||cfdata==28||cfdata==29||cfdata==31){
cfout=4;
}
}
else{
cfout=7;
}
return cfout;
}
int MOTOR_FUNC_A(int cfina)
{
int aout=0,cfadata=cfina;
int mask_addra=7;
mask_addra&=cfina;
if(mask_addra==4||mask_addra==2){
cfadata ≫=3;
if (cfadata== 20|| cfadata == 21|| cfadata ==23){
aout= pos;
} else if(cfadata== 28|| cfadata == 29|| cfadata ==31){
aout=neg;
} else if(cfadata ==5 ||cfadata ==7){
aout = Z;
}
}
else{
aout = error;
}
return aout;
}
int MOTOR_FUNC_B(int cfinb)
{ int bout=0, cfbdata=cfinb;
int mask_addrb=7;
mask_addrb&=cfinb;
if(mask_addrb==4||mask_addrb==2){
cfbdata ≫=3;
```

```
if (cfbdata== 5|| cfbdata == 21|| cfbdata ==29){
bout= pos;
} else if(cfbdata== 7|| cfbdata == 23|| cfbdata ==31){
bout=neg;
} else if(cfbdata== 20|| cfbdata == 28){
bout = Z;
}
}
else{
bout = error;
}
return bout;
}
/*Serial to Parallel converter */
int Rx(int inbit, int Prev_Frame)
{
int Frame_In=Prev_Frame;
int Mask_Msb = 256;
/* assuming LSB first transmission*/
if (inbit==0){
Frame_In ≫= 1;
} else if(inbit==1){
Frame_In —= Mask_Msb;
Frame_In ≫= 1;
/* cout ≪"\n after shifting once " ≪ Frame_In ≪ "\ n";
/
}
/* else{
cout ≪ "\n input error \n";
}*/ return Frame_In;
} int Get_push(int push_seq, int step_count)
{ const int step_array[8]={168,160,184, 56, 248, 224, 232, 40};
int push_val, index;
if(push_seq==1)
index=(step_count+2)%8;
/* fcw */
else if(push_seq==2)
index = (step_count%8)+1;
/* hcw */
else if(push_seq==3)
index = (step_count+6)%8;
/* fccw */
```

```
else
index = (step_count+6)%8+1;
/* hccw */
push_val= step_array[(index-1)];
return push_val;
}
int Next_step(int push_seq, int step_count)
{
int index;
if(push_seq==1)
index=(step_count+2)%8;
/* fcw */
else if(push_seq==2)
index = (step_count%8)+1;
/* hcw */
else if(push_seq==3)
index = (step_count+6)%8;
/* fccw */
else
index = (step_count+6)%8+1;
/* hccw */
return index;
}
```

```
* Assembly code listing
* file of standard 6811 register declarations
**********************************************************
Control Registers
BASE EQU $1000
PORTA EQU $1000 ; Port A data register
RESV1 EQU $1001 ; Reserved
PIOC EQU $1002 ; Parallel I/O Control register
PORTC EQU $1003 ; Port C latched data register
PORTB EQU $1004 ; Port B data register
PORTCL EQU $1005 ;
DDRC EQU $1007 ; Data Direction register for port C
PORTD EQU $1008 ; Port D data register
DDRD EQU $1009 ; Data Direction register for port D
PORTE EQU $100A ; Port E data register
CFORC EQU $100B ; Timer Compare Force Register
OC1M EQU $100C ; Output Compare 1 Mask register
OC1D EQU $100D ; Output Compare 1 Data register
* Two-Byte Registers (High,Low - Use Load & Store Double to access)
TCNT EQU $100E ; Timer Count Register
TIC1 EQU $1010 ; Timer Input Capture register 1
TIC2 EQU $1012 ; Timer Input Capture register 2
TIC3 EQU $1014 ; Timer Input Capture register 3
TOC1 EQU $1016 ; Timer Output Compare register 1
TOC2 EQU $1018 ; Timer Output Compare register 2
TOC3 EQU $101A ; Timer Output Compare register 3
TOC4 EQU $101C ; Timer Output Compare register 4
TI4O5 EQU $101E ; Timer Input compare 4 or Output compare 5 register
TCTL1 EQU $1020 ; Timer Control register 1
TCTL2 EQU $1021 ; Timer Control register 2
TMSK1 EQU $1022 ; main Timer interrupt Mask register 1
TFLG1 EQU $1023 ; main Timer interrupt Flag register 1
TMSK2 EQU $1024 ; misc Timer interrupt Mask register 2
TFLG2 EQU $1025 ; misc Timer interrupt Flag register 2
PACTL EQU $1026 ; Pulse Accumulator Control register
PACNT EQU $1027 ; Pulse Accumulator Count register
SPCR EQU $1028 ; SPI Control Register
SPSR EQU $1029 ; SPI Status Register
SPDR EQU $102A ; SPI Data Register
BAUD EQU $102B ; SCI Baud Rate Control Register
SCCR1 EQU $102C ; SCI Control Register 1
SCCR2 EQU $102D ; SCI Control Register 2
```

```
SCSR EQU $102E ; SCI Status Register
SCDR EQU $102F ; SCI Data Register
ADCTL EQU $1030 ; A/D Control/status Register
ADR1 EQU $1031 ; A/D Result Register 1
ADR2 EQU $1032 ; A/D Result Register 2
ADR3 EQU $1033 ; A/D Result Register 3
ADR4 EQU $1034 ; A/D Result Register 4
BPROT EQU $1035 ; Block Protect register
RESV2 EQU $1036 ; Reserved
RESV3 EQU $1037 ; Reserved
RESV4 EQU $1038 ; Reserved
OPTION EQU $1039 ; system configuration Options
COPRST EQU $103A ; Arm/Reset COP timer circuitry
PPROG EQU $103B ; EEPROM Programming register
HPRIO EQU $103C ; Highest Priority Interrupt and misc.
INIT EQU $103D ; RAM and I/O Mapping Register
TEST1 EQU $103E ; factory Test register
CONFIG EQU $103F ; Configuration Control Register
* Interrupt Vector locations
SCIINT EQU $D6 ; SCI serial system
SPIINT EQU $D8 ; SPI serial system
PAIINT EQU $DA ; Pulse Accumulator Input Edge
PAOVINT EQU $DC ; Pulse Accumulator Overflow
TOINT EQU $DE ; Timer Overflow
TOC5INT EQU $E0 ; Timer Output Compare 5
TOC4INT EQU $E2 ; Timer Output Compare 4
TOC3INT EQU $E4 ; Timer Output Compare 3
TOC2INT EQU $E6 ; Timer Output Compare 2
TOC1INT EQU $E8 ; Timer Output Compare 1
TIC3INT EQU $EA ; Timer Input Capture 3
TIC2INT EQU $EC ; Timer Input Capture 2
TIC1INT EQU $EE ; Timer Input Capture 1
RTIINT EQU $F0 ; Real Time Interrupt
IRQINT EQU $F2 ; IRQ External Interrupt
XIRQINT EQU $F4 ; XIRQ External Interrupt
SWIINT EQU $F6 ; Software Interrupt
BADOPINT EQU $F8 ; Illegal Opcode Trap Interrupt
NOCOPINT EQU $FA ; COP Failure (Reset)
CMEINT EQU $FC ; COP Clock Monitor Fail (Reset)
RESETINT EQU $FE ; RESET Interrupt
************************************************
ORG MAIN_START
```

```
addr_sel FCB 0
dir_sel FCB 0
step_sel FCB 0
curr_step FCB 0
next_step FCB 0
motor_mode FCB 0 ; 1,2,4,8 for fcw,fccw,hcw,hccw
variable_program_status FDB 0
pull_val FCB 0
prog_count FCB 0
ovr_count FCB 0 ; see initialization module
ovr_step FCB 0 ; see init.. module
push_begin FCB $00,$A8,$A0,$B8,$38,$F8,$E0,$E8,$28; seq for half cw
push_frame FCB 0
toc_val FCB 0 ; timer int address variable
prog_status FCB 0
info_count FCB 100
subroutine_initialize_module:
*variables to be initialized
LDAA #10
STAA toc_val ; the timer init routine
LDAA #4
STAA ovr_count
LDAA #8
STAA ovr_step
LDX #BASE
LDAA #$3C
STAA DDRD,X ; make SPI pins outputs
BCLR PORTD,X $3C
LDAA #$80
STAA PACTL,X
; enable PA7 for output
BCLR PORTA,X $80
*******************************************************
* File "ldxibase.asm"
* Fred Martin Thu Oct 10 19:49:38 1991
* The following code loads the X register with a base pointer to
the 6811 interrupt vectors: $FF00 if the 6811 is in normal mode,
and $BF00 if the 6811 is in special mode.
* The file "6811regs.asm" must be loaded first for this to work.
LDAA HPRIO
ANDA #$40 ; test SMOD bit
BNE *+7
```

```
LDX #$FF00 ; normal mode interrupts
BRA *+5
LDX #$BF00 ; special mode interrupts
*********************************************************
LDD #toc3_int
STD TOC3INT,X
LDX #BASE
LDD TCNT,X ; timer initialization
ADDD #10000
STD TOC3,X
LDAA #%00100000
STAA TFLG1,X
STAA TMSK1,X ; enable timer3 interrupt
LDAA #%00010000
STAA TCTL1,X ; test pa5 with this for 5msec waveform
CLI
RTS
*********************************************************
Interrupt service routine:
*********************************************************
toc3_int:
LDAA #10
CMPA toc_val
BNE toc3_mid0
BSR start_timer5
BRA timer_0 ; nearest timer also if toc_val=70
*********************************************************
start_timer5:
*********************************************************
init section
*********************************************************
LDX #BASE
LDAA #1
CMPA prog_status
BEQ go_normal ; device programmed-move to normal operation
BSET PORTA,X $80 ; set PA7 to 1 indicating programming
wait_prog:
LDAA $7FFF
ANDA #$04 ; testing d12 from user for programming
BNE wait_prog
BSET PORTD,X $10 ; set PD4 (PA7) is already high(processing)
BSR set_addr
```

```
BSET PORTD,X $04 ; enable PD2 for 12volt
LDAB addr_sel ; this is the addr to be progmed
STAB push_frame ;
LDAA #2 ; first two steps of programming
STAA prog_count
LDAA #20 ; arbit,programming start
STAA toc_val ; toc int serviced by programming
RTS
go_normal:
wait_normal:
LDAA $7FFF
ANDA #$04
BEQ wait_normal ; await d12 disable
BSET PORTD,X $08 ; set PD3,and clr PD4 and PA7 for OK code
BCLR PORTD,X $10
BCLR PORTA,X $80
BRSET $7FFF $10 dir_cw ; otherwise, dir remains 0
step_set: BRSET $7FFF $20 step_full ; otherwise,step remains half
addr_set: BSR set_addr
BSR set_push ; load the next push sequence
LDAA #60
STAA toc_val ; toc int serviced by normal loop
RTS
set_addr:
BRCLR $7FFF $08 addr_other ; testing d13 from user for address
LDAB #4 ; this is 001 in reverse a0a1a2
STAB addr_sel
RTS
addr_other:
LDAB #2 ; the address is 010
STAB addr_sel
RTS
dir_cw:
LDAA #1 ; setting direction to 1
STAA dir_sel
BRA step_set
step_full:
LDAA #1 ; setting step to full
STAA step_sel
BRA addr_set
set_push:
TST step_sel
```

```
BNE full_dir_chk
half_dir_chk:
TST dir_sel
BNE half_cw
BRA half_ccw_mid
**********out range branch
toc3_mid0:
BRA toc3_mid1
timer_0:
BRA timer_1
full_dir_chk:
TST dir_sel
BNE full_cw
BRA full_ccw
full_cw:
LDAA #1
STAA motor_mode
****
reqd steps are 1,3,5,7,1
*********
fcw:
LDAA curr_step
BEQ add_1
CMPA #7
BEQ add_1
CMPA #1
BEQ add_3
CMPA #3
BEQ add_5
BRA add_7
**************************out of range branch
set_push_mid5:
BRA set_push
************************************************
full_ccw:
LDAA #2
STAA motor_mode
fccw:
LDAA curr_step
BEQ add_7 ; reqd steps are 7,5,3,1,7,..
CMPA #1
BEQ add_7
```

```
CMPA #7
BEQ add_5
CMPA #5
BEQ add_3
BRA add_1
**********out of range branch
half_ccw_mid:
BRA half_ccw
**********
half_cw:
LDAA #4
STAA motor_mode
hcw: LDAA curr_step
BEQ add_1
; reqd steps are 1,2,....8,1
CMPA #8
BEQ add_1
CMPA #1
BEQ add_2
CMPA #2
BEQ add_3
CMPA #3
BEQ add_4
CMPA #4
BEQ add_5
CMPA #5
BEQ add_6
CMPA #6
BEQ add_7
BEQ add_8
**********out of range branch
fcw_mid5:
BRA fcw
fccw_mid5:
BRA fccw
set_push_mid4:
BRA set_push_mid5
hcw_mid5:
BRA hcw
timer_1:
BRA timer_1_mi
d toc3_mid1:
```

```
BRA toc3_mid
**************************************************
add_1:
LDAA #1
LDAB push_begin+1
BRA push_val
add_2:
LDAA #2
LDAB push_begin+2
BRA push_val
add_3:
LDAA #3
LDAB push_begin+3
BRA push_val
add_4:
LDAA #4
LDAB push_begin+4
BRA push_val
add_5:
LDAA #5
LDAB push_begin+5
BRA push_val
add_6:
LDAA #6
LDAB push_begin+6
BRA push_val
add_7:
LDAA #7
LDAB push_begin+7
BRA push_val
add_8:
LDAA #8
LDAB push_begin+8
BRA push_val
push_val:
STAA next_step
ORAB addr_sel ; push frame=data+address
STAB push_frame ; this is the push data
RTS
timer_1_mid:
BRA timer_init0
half_ccw:
```

```
LDAA #8
STAA motor_mode
hccw:
LDAA curr_step
BEQ add_8 ; reqd steps are 8,7,.....1,8
CMPA #1
BEQ add_8
CMPA #2
BEQ add_1
CMPA #3
BEQ add_2
CMPA #4
BEQ add_3
CMPA #5
BEQ add_4
CMPA #6
BEQ add_5
CMPA #7
BEQ add_6
BEQ add_7
**************************************************
out of range branches
**************************************************
fcw_mid4:
BRA fcw_mid5
fccw_mid4:
BRA fccw_mid5
set_push_mid3:
BRA set_push_mid4
hcw_mid4:
BRA hcw_mid5
hccw_mid4:
BRA hccw
**************************************************
end of init section
**************************************************
toc3_mid:
BSR push_seq_mid0 ; nearest push_seq
LDAA #20 ; this is the pull analysis part
CMPA toc_val
BEQ step_0
LDAA #30
```

```
CMPA toc_val
BEQ step_3
LDAA #40
CMPA toc_val
BEQ step_4
LDAA #50
CMPA toc_val
BEQ step_5_mid
LDAA #60
CMPA toc_val
BEQ push_normal1
BRA timer_init0 ; nearest timer also if toc_val=70
*****************************************************
step_0
LDX #BASE
LDAA #6 ; check for 110 code
CMPA pull_val
BNE TI_step0 ; repeat until 110
BSET PORTD,X $08 ; set PD3
BSET PORTD,X $10 ; set PD4
BCLR PORTD,X $80
DEC prog_count
BEQ TI_step3 ; proceed with next step
BRA TI_step1_2
TI_step0:
LDAA #2 ; first two steps of programming
STAA prog_count
TI_step1_2:
LDAA #20 ; the next step is step_0
STAA toc_val
BRA timer_init1
*********************************************************
out of range branches
*********************************************************
timer_init0:
BRA timer_init1
push_seq_mid0:
BRA push_seq_mid
set_push_mid2:
BRA set_push_mid3
fcw_mid3:
BRA fcw_mid4
```

```
fccw_mid3:
BRA fccw_mid4
hcw_mid3:
BRA hcw_mid4
hccw_mid3:
BRA hccw_mid4
step_5_mid:
BRA step_5
****************************************
TI_step3:
BCLR PORTD,X $04 ; disable PD2 for 5volts
LDAA #30
STAA toc_val
BRA timer_init
1
step_3:
LDAA #4
CMPA pull_val
BNE TI_step0 ; if 100 code,proceed to ovrbit program
TI_init_step4:
BSET PORTD,X $04 ; enable PD2 for 12volts
LDAB addr_sel
ADDB $08 ; setting D4 to 1 for ovrbit program
STAB push_frame ; new push for ovrbit
TI_step4:
LDAA #40 ; from second time onwards
STAA toc_val
BRA timer_init1
step_4:
DEC ovr_step
LDAA #6
CMPA pull_val
BNE TI_step5 ; code is incorrect
INC ovr_count ; correct code count for ovrbitprogram
BRA TI_step5
******************
out of range branches
******************
push_normal1:
BRA push_normal2
set_push_mid1:
BRA set_push_mid2
```

```
push_seq_mid:
BRA push_seq_mid1
timer_init1:
BRA timer_init
hccw_mid2
: BRA hccw_mid3
hcw_mid2:
BRA hcw_mid3
fccw_mid2:
BRA fccw_mid3
fcw_mid2:
BRA fcw_mid3
*****************************************
TI_step5:
LDAA #50
STAA toc_val
BRA timer_init
step_5:
DEC ovr_step
LDAA #4
CMPA pull_val
BNE count_skip
INC ovr_count
count_skip:
LDAA ovr_count
CMPA ovr_step
BEQ TI_normal ; we are done with programming
err_chk:
TST ovr_step ; try 4 times before error service
BNE TI_step4
BRA error_service
*******************
out of range branches
*******************
push_seq_mid1:
BRA push_seq
*********************
TI_normal:
LDX #BASE
BCLR PORTD,X $04 ; clear PD2
BCLR PORTD,X $08 ; clear PD3- programming done.
BSET PORTD,X $10 ; set PD4 ok code
```

```
BSR set_push_mid1 ; get normal push value
LDAA #60
STAA toc_val
BRA timer_init
error_service:
BSET PORTD,X $18 ; set PD3, PD4 - for error code
LDAA #70 ; do nothing from now on. reset reqd
STAA toc_val
BRA timer_init
***********
out of range branches
***************
push_normal2:
BRA push_normal3
timer_init:
BRA timer_init2
hccw_mid1:
BRA hccw_mid2
hcw_mid1:
BRA hcw_mid2
fccw_mid1:
BRA fccw_mid2
RTS
fcw_mid1:
BRA fcw_mid2
*****************************************
push_seq:
*send bus violation
BSR bus_v3
send push sync
BSR push_sync
send push field
BSR push_8
send pull sync and receive pull_data
BSR pull_3
RTS
bus_v3:
LDY #3
bus_n:
BSR delay1_25
BCLR PORTD,X $20 ; PD5 push bit
BSR delay0_25
```

```
BSET PORTD,X $20
BSR delay1_25 ; smaller delay
DEY
BNE bus_n
RTS
push_sync:
BCLR PORTD,X $20 ; PD5 push bit
BSR delay0_25
NOP
BSET PORTD,X $20
BSR delay0_25
NOP
BSET PORTD,X $20
BSR delay0_25
NOP
BCLR PORTD,X $20
BSR delay1_25
NOP
RTS
push_8:
LDY #8
LDAB push_frame
push_n LSLB
BCC goto_0
BRA goto_1
goto_0:
BSR code_1
BSR code_0
DEY
BNE push_n
RTS
goto_1:
BSR code_0
BSR code_1
DEY
BNE push_n
RTS
code_0:
BCLR PORTD,X $20
BSR delay0_25
BSET PORTD,X $20
NOP
```

```
RTS
code_1:
BSET PORTD,X $20
BSR delay0_25
BCLR PORTD,X $20
NOP
RTS
************
out of range branches
***************
push_normal3:
BRA push_normal
timer_init2:
BRA timer_init3
hccw_mid0:
BRA hccw_mid1
hcw_mid0:
BRA hcw_mid1
fccw_mid0:
BRA fccw_mid1
fcw_mid0:
BRA fcw_mid1
push_seq_mid2:
BSR push_seq
RTS
****************************************
delay_75:
LDAA #22
loop1 DECA
BNE loop1
RTS
delay0_25:
LDAA #1
loop2 DECA
BNE loop2
RTS
delay1_25:
LDAA #3
loopd DECA
BNE loopd
RTS
```

```
pull_3:
BSR send_1 ; sending pull sync bit
BSR delay0_25
BSR code_0
BSR delay1_25
BSR code_1
BSR delay1_25
BSET PORTD,X $20
LDY #0003 ; we read 3 bits
pull_next BSR delay_15 ; trying to read pull bit after this delay
LDAA PORTA,X ; move this label to prev line if delay rqd
ANDA #%00000001 ; test PA0 pull bit
BEQ pull_0 ; the bit is zero
pull_1: LDAA pull_val
ORAA %00000001
STAA pull_val
pull_0: LSL pull_val
DEY
BNE pull_next
pull_eof:
LDAA PORTA,X ; we read end of frame. only delay is diff.
pull_eof2 BSR delay0_25 ; eof has 50micro pulse(20khz)
BSR delay_15
LDAB PORTA,X
SBA
BEQ err_eof ; if both bits are same, this is an error
RTS
err_eof BSET PORTD,X $10 ; set PA7, PD4 - for error code
BSET PORTA,X $80
LDAA #70 ; do nothing from now on. reset reqd
STAA toc_val
BRA timer_init_mid
delay_15:
LDAA #1
loop3 DECA
BNE loop3
RTS
************
out of range branches
***************
timer_init3:
```

```
BRA timer_init_mid
************************
hccw_mid:
BRA hccw_mid0
hcw_mid:
BRA hcw_mid0
fccw_mid:
BRA fccw_mid0
fcw_mid:
BRA fcw_mid0
****************************************
push_normal:
LDAA #4
CMPA pull_val ; check whether status is ok code
BNE push_set ; repeat prev step if not OK
TI_next_normal
LDAA #1
CMPA motor_mode
BEQ mode_1
LSLA
CMPA motor_mode
BEQ mode_2
LSLA
CMPA motor_mode
BEQ mode_3
BRA mode_4
mode_1 BSR fcw_mid
BRA push_set
mode_2 BSR fccw_mid
BRA push_set
mode_3 BSR hcw_mid
BRA push_set
mode_4 BSR hccw_mid
push_set:
LDAA #60 ; next push val
STAA toc_val
LDAA next_step
STAA curr_step ; overwrite currstep with the new step
BRA timer_init_mid
timer_init_mid:
no_info:
LDX #BASE
```

```
LDD TCNT,X
ADDD #5000
STD TOC3,X
BCLR TFLG1,X %11011111 ; this will work only if the prev
RTI ; part takes less than 5msec..
```

Ptolemy Schematics

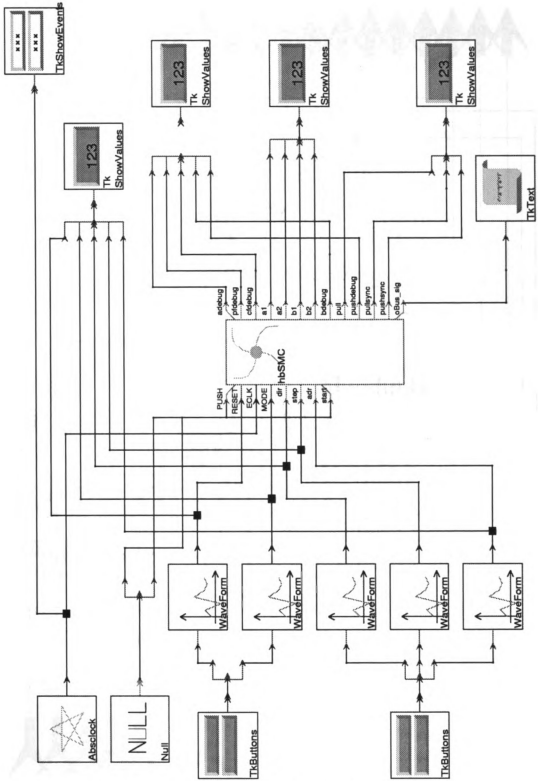


Figure 7.1: The SMC Testbed Universe

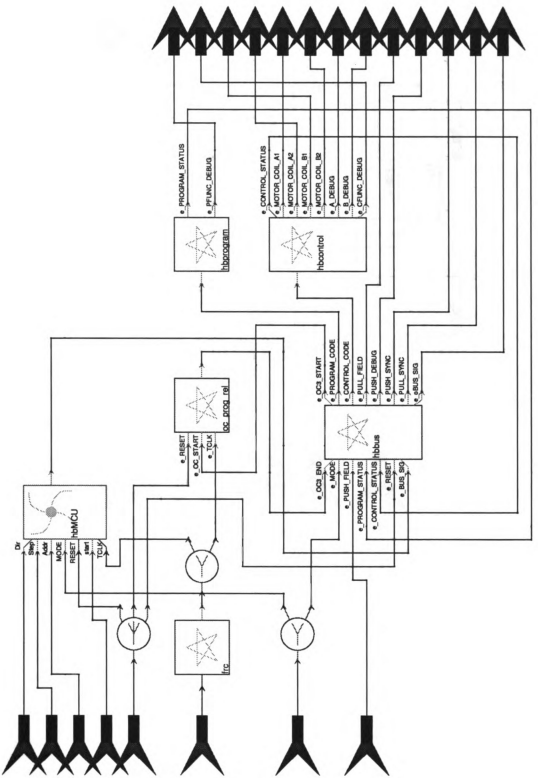


Figure 7.2: The "SMC" Galaxy

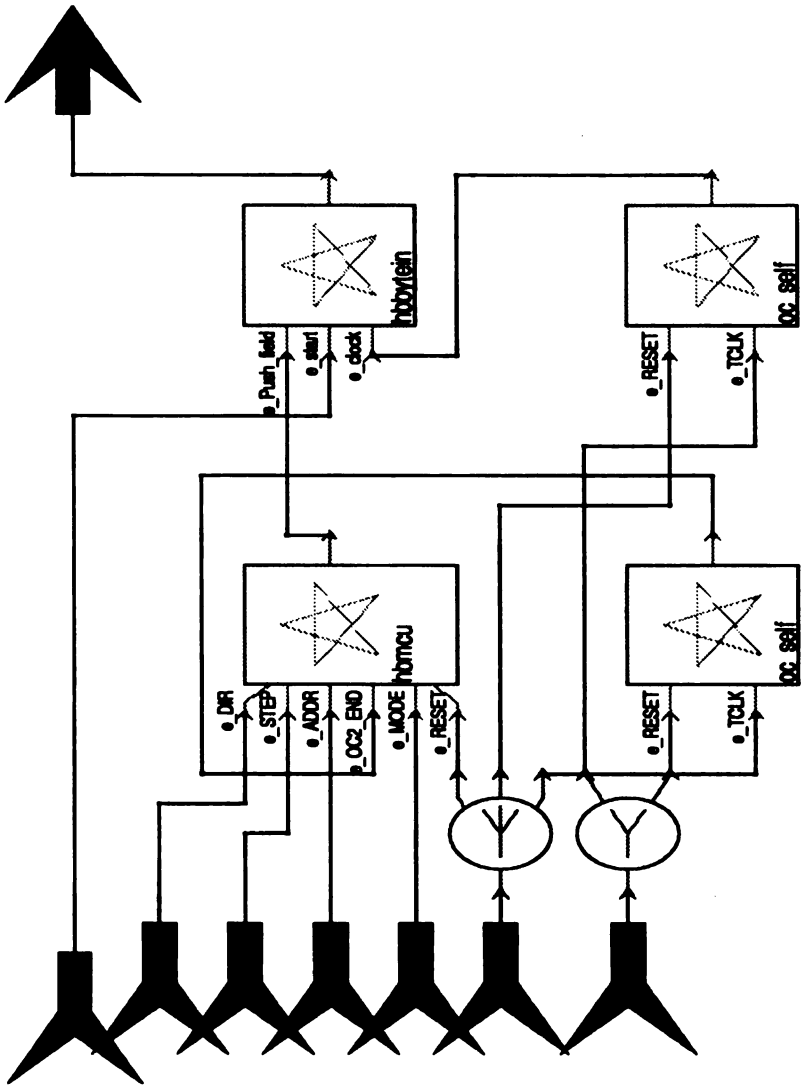


Figure 7.3: The "mcu" Galaxy

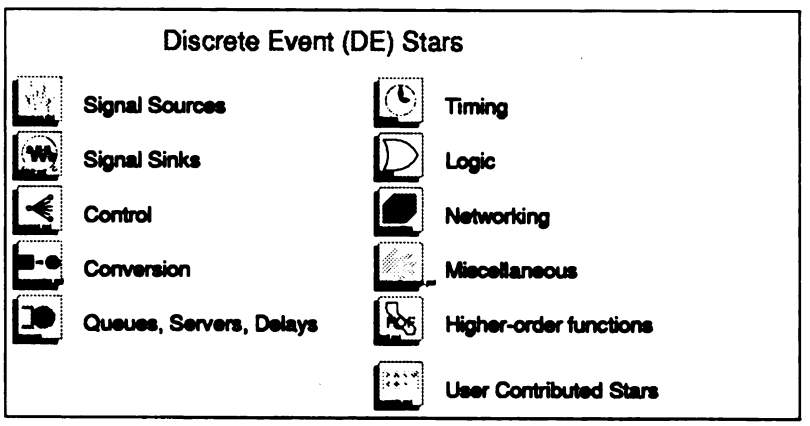


Figure 7.4: Ptolemy Icons

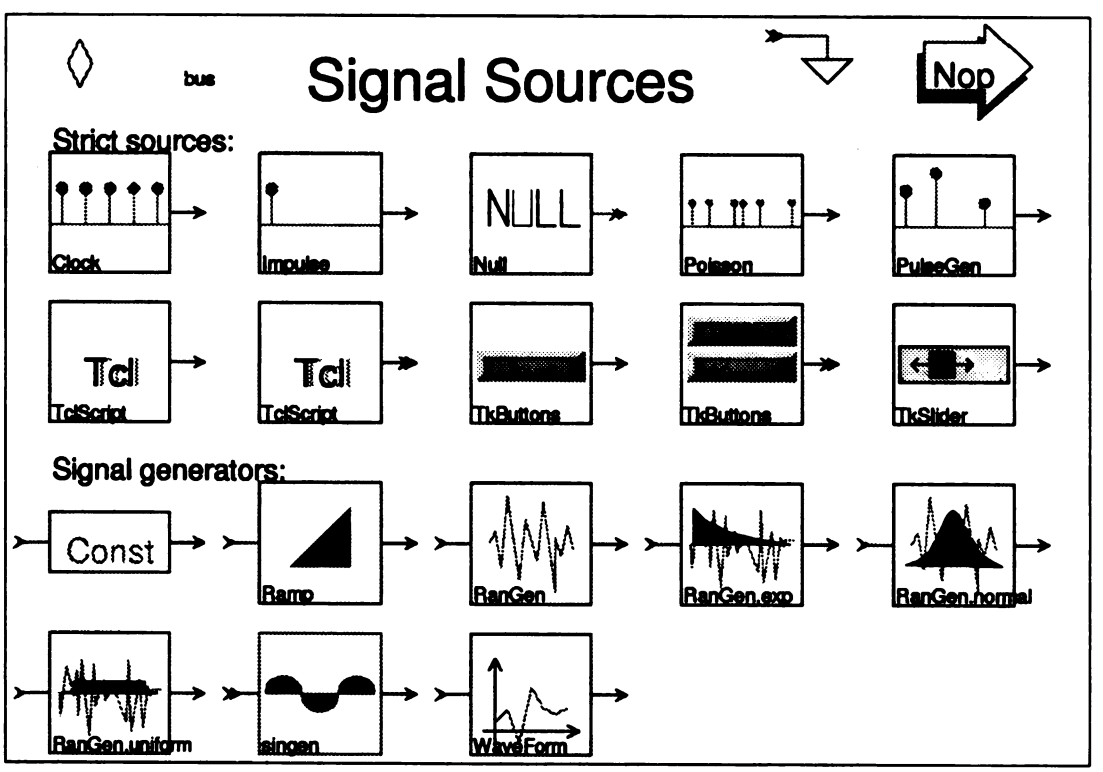


Figure 7.5: Ptolemy Icons - Sources

Figure 7.6: Universe Parameters

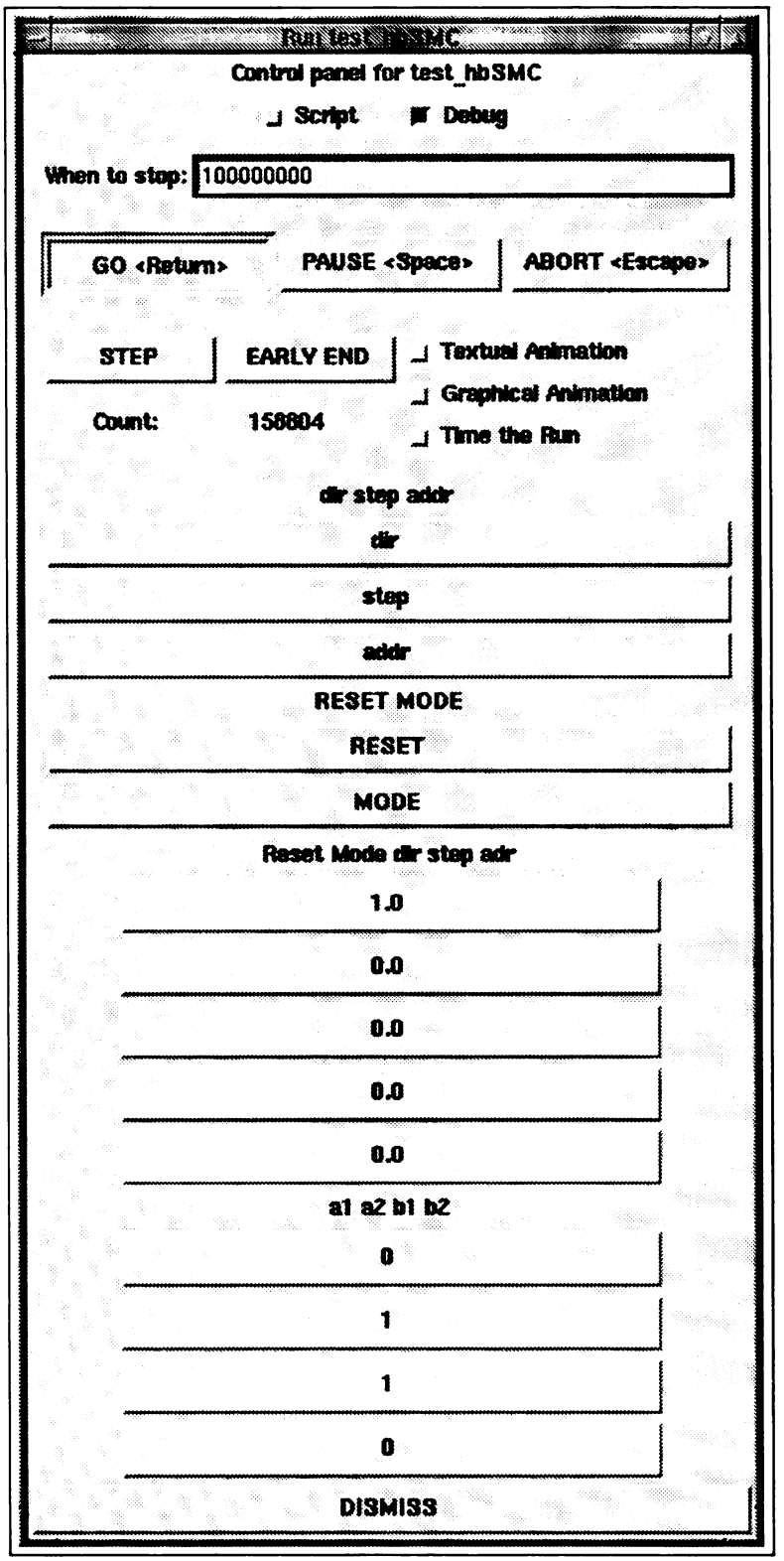


Figure 7.7: The SMC Control Panel

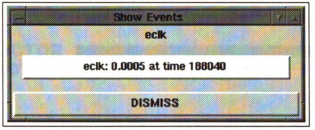


Figure 7.8: The SMC eclk

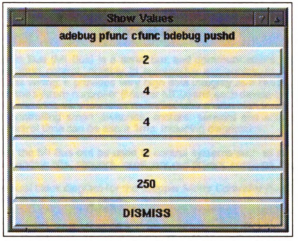


Figure 7.9: The SMC Event Display

Order this document
as AN475/D

MOTOROLA
SEMICONDUCTOR
APPLICATION NOTE

AN475

Single wire MI Bus controlling stepper motors

by Michel Burri & Dr. Pascal Renard
Senior Staff engineers, Automotive Group
Motorola SA, Geneva

1. Introduction

The Motorola Interconnect Bus (MI Bus) is a serial bus and communications protocol which efficiently supports distributed real time control, notably in automotive electronics. In addition to being a cost-effective alternative to bulk wiring, it provides very high data integrity as a result of continuous Push-Pull communication between the system controller (Master MCU) and each device on the bus. It is suitable for medium speed networks requiring very low cost multiplex wiring with high levels of noise immunity. The MI Bus is suitable for controlling smart switches, motors, sensors and actuators with a single-chip controller. The process control time can be about 1ms, including diagnostics.

In automotive electronics the MI Bus can be used to control systems such as air conditioning, head light levellers, mirrors, seats, window lifts, sensors, intelligent coil drivers, consoles, dashboard etc.

Figure 1-1 shows the general block diagram for the Stepper Motor Controller (SMC). The main parts of the diagram will be discussed in the following pages.

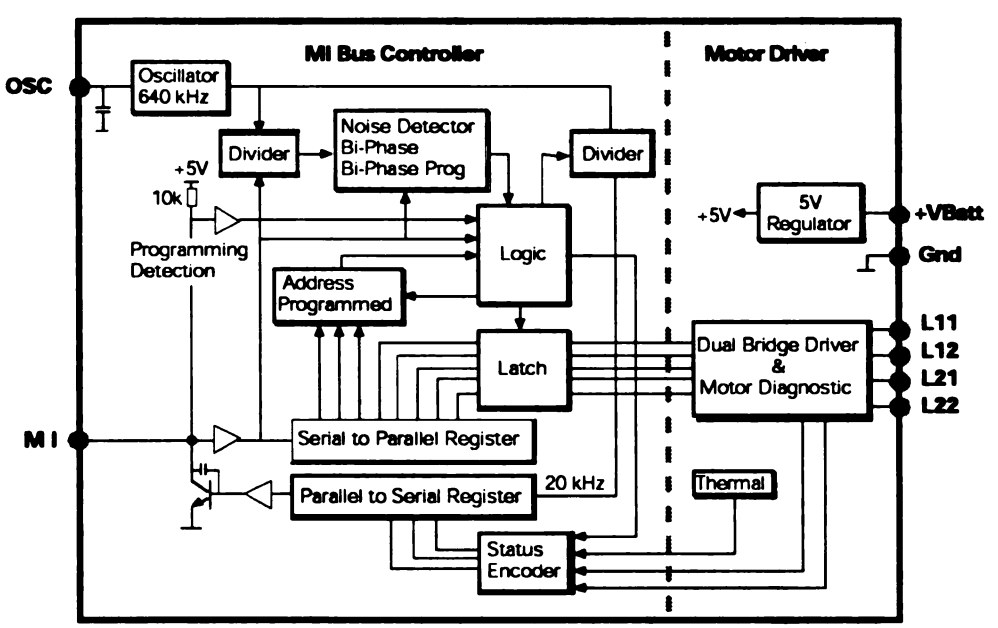


Figure 1-1 Stepper motor controller block diagram

MAXIMUM RATINGS (All voltages are with respect to ground, unless otherwise noted.)

Rating	Symbol	Value	Limit
Power Supply Voltage Continuous Operation Transient Survival (Note 1)	 V_{CC} V_{LD}	 25 40	V
Digital Input Voltage	V_I	0.3 to V_{CC} + 0.3	V
Output Current (T_A = – 40°C)	I_{OLT}	260	mA
Output Current (T_A = 100°C)	I_{OHT}	150	mA
Storage Temperature	T_{stg}	– 40 to +150	°C
Operating Temperature (Note 2)	T_A	– 40 to +125	°C
Junction Temperature	T_J	– 40 to +150	°C
Power Dissipation (T_A = 100°C)	P_D	0.6	W
Load Dump Transient (Note 3)	V_{LD}	40	V

DC ELECTRICAL CHARACTERISTICS (Characteristics noted under conditions 9.0 V ≤ V_{CC} ≤ 15.5 V, – 40°C ≤ T_A ≤ 100°C, unless otherwise noted.)

Characteristic	Symbol	Min	Typ	Max	Unit
Standby Current (V_{CC} = 15.5 V) (Note 4)	I_Q	–	–	12	mA
Output Current (V_{CC} = 15.5 V)	I_O	–	120	–	mA
H–Bridge Saturation Voltage (I_O = 150 mA) (Note 5) T_A = – 40°C T_A = 25°C T_A = 100°C	$V_{O(sat)}$	– – – –	– 1.3 1.2 1.1	– 1.6 1.6 1.6	V
Address Programming Current (T_A = 25°C) (Note 6)	I_{pc}	–	1.2	–	A

CONTROL LOGIC ELECTRICAL CHARACTERISTICS (Characteristics noted under conditions 9.0 V ≤ V_{CC} ≤ 15.5 V, – 40°C ≤ T_A ≤ 100°C, unless otherwise noted.)

Characteristic	Symbol	Min	Typ	Max	Unit
Oscillator (Note 7)	f_{cl}		640	–	kHz
Message Time Slot (V_{CC} = 12 V) (Note 8)	t_s	24.8	25	25.2	µs
Urgent Output Disable (V_{CC} = 12 V) (Note 9)	t_{od}	9 x t_s	–	–	µs
Internal MI–Bus Pull–Up Resistor	R_{pu}	6.0	–	20	kΩ
Internal MI–Bus Zener Diode Clamp Voltage	V_{cl}	–	18	–	V
Address Programming Voltage (Note 10)	V_p	10	12	14	V
Program Energize Time	t_{ppw}	200		1000	µs
MI–Bus Slew Rate	$\Delta V/\Delta t$	1.0	1.5	2.0	V/µs
MI–Bus "0" Level Input Voltage Threshold	V_{il}	–	–	1.3	V
MI–Bus "1" Level Input Voltage Threshold	V_{ih}	2.4	–	–	V
MI–Bus "0" Level Output Voltage (I_O = 30 mA)	V_{OL}	–	–	1.0	V
Power–On Reset Time (V_{CC} ≥ 7.5 V)	t_{por}	–	250	–	µs

NOTES: 1. Transient capability is defined as the positive overvoltage transient with 250 ms decay time constant. The detection on an overvoltage condition causes all H–Bridges to be latched "off".
2. Ambient temperature is given as a convience; Maximum junction temperature is the limiting factor.
3. Load Dump is the inductive transient voltage imposed on an automotive battery line as a result of opening the battery connection while the alternator system is producing charge current. The detection on an overvoltage condition causes all H–Bridges to be latched "off".
4. Standby Current is with both H–Bridges "off" (Inh1 = Inh2 = 0).
5. H–Bridge Saturation Voltage is referenced to the positive supply or ground respective of the H–Bridge output being High or Low. Saturation voltage is the voltage drop from the output to the positive supply (with output High) and the voltage drop to ground (with output Low).
6. Address Programming Current is the current encountered when the bus is at 12 V during address programming.
7. A typical application uses an external ceramic resonator crystal having a frequency of 644 kHz. An internal capacitor in parallel with ceramic resonator is used to shift the frequency to the working frequency of 640 kHz. The frequency accuracy of the oscillator is dependant on the capacitor and ceramic resonator tolerance (usually ±1.0%).
8. The Message Time Slot is the time required for one complete device message transfer. The message time is equivalent to a total of 16 periods of the oscillator frequency used.
9. If the MI–Bus becomes shorted to ground, all MC33192 outputs will be disabled after a period of nine time slots ($9t_s$).
10. MI–Bus voltage required for address programming.

GENERAL DESCRIPTION

The MC33192 is a serial stepper motor controller for use in harsh automotive applications using multiplex wiring. The MC33192 provides all the necessary four phase drive signals to control two phase bipolar stepper motors operated in either half or full step modes. Multiple stepper motor controllers can be operated on a real time basis at step frequencies up to 200 Hz using a single microcontroller (MCU). A primary attribute of operation is the utilization of the MI–Bus message media to provide high noise immunity communication ensuring very high operating reliability of motor stepping.

The MC33192 is designed to drive bipolar stepper motors having a winding resistance of 80 Ω at 20°C with a supply voltage of 12 V. It is supplied in a SO–16L plastic package having eight pins, on one side, connected directly to the lead frame thus enhancing the thermal performance to allow a power dissipation of 0.5 W at 120°C ambient temperature.

Multiple Simultaneous Motor Operation

Several motors can be controlled in a serial fashion, one after the other, using the same software time base. The time base determines the step frequency of the motors. A single motor can be operated at a maximum speed of 200 Hz pull–in with a duration of 5.0 ms per step. Three motors can be operated simultaneously using a 68HC05B6 MCU at the same time base (200 Hz) with about 1.7 ms per step. A 68HC11 MCU can control 4 stepper motors with adequate program step time. The step frequency must be decreased to control additional motors. To control eight motors simultaneously would require the motor speed to be decreased to 100 Hz producing about 2.0 ms time duration per step with adequate program time.

MI–Bus General Description

The Motorola Interconnect Bus (MI–Bus) is a serial push–pull communications protocol which efficiently supports distributed real time control while exhibiting a high level of noise immunity.

Under the SAE Vehicle Network categories, the MI–Bus is a Class A bus with a data stream transfer bit rate in excess of 20 kHz and thus inaudible to the human ear. It requires a single wire to carry the control data between the master MCU and its slave devices. The bus can be operated at lengths up to 15 meters.

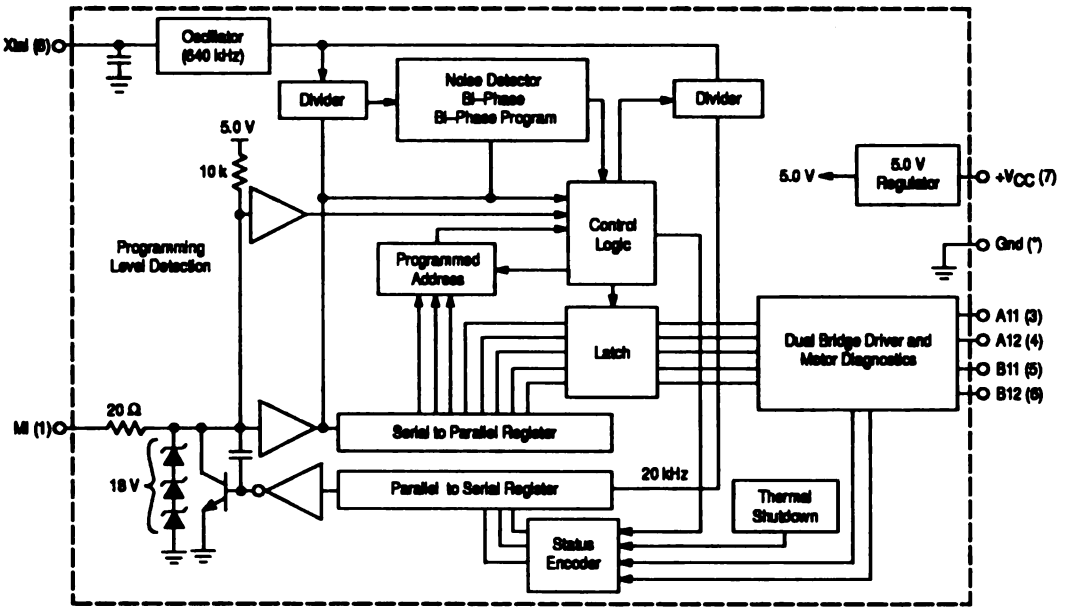
At 20 kHz the time slot used to construct the message (25 µs) can be handled by software using many MCUs available on the market.

The MI–Bus is suitable for medium speed networks requiring very low cost multiplex wiring. Aside from ground, the MI–Bus requires only one signal wire connecting the MCU to multiple slave MC33192 devices with individual control.

A single MI–Bus can accomplish simultaneous automotive system control of Air Conditioning, Head Lamp Levellers, Window Lifts, Sensors, Intelligent Coil Drivers, etc. The MI–Bus has been found to be cost effective in vehicle body electronics by replacing the conventional wiring harness.

Figure 1 shows the internal block diagram of the MC33192 Stepper Motor Controller.

Figure 1. MC33192 Stepper Motor Controller Block Diagram



NOTE: (*) Pins 2, 9, 10, 11, 12, 13, 14, 15 and 16 are common electrical and heatsink ground pins for the device.

MI–Bus Access Method

The information on the MI–Bus is sent in a fixed message frame format (See Figure 4). The system MCU can take control of the MI–Bus at any time with a start bit which violates the law of Manchester Bi–Phase code by having three consecutive Time Slots ($3t_B$) held constantly at a Logic "0" state.

Push–Pull Communication Sequence

Communication between the system MCU and slave MC33192 devices always use the same message frame organization. The MCU first sends eight serial data bits over the MI–Bus comprised of five control bits followed by three address bits. This communication sequence is called a "Push Field" since it represents command information sent from the MCU. The sequence of the five control data bits follow the order D0, D1, D2, D3 and D4. The three address bits are sent in sequential order A0, A1 and A2 defining a binary address code. The condition of MI–Bus during any of the control bit time windows defines a specific control function as shown in Figure 2. A "Pull Sync" bit is sent at the end of the Push Field, the positive edge of which causes all data sent to the selected device to be latched into the output circuit.

Figure 2. Push Field Data Bits

Bit	Name	Control Function
D4	Inh2	Inhibits H–Bridge 2
D3	Dir2	Establishes Direction of H–Bridge 2 Current
D2	E	Energizes Bridge Coils 1 and 2
D1	Dir1	Establishes Direction of H–Bridge 1 Current
D0	Inh1	Inhibits H–Bridge 1

After the Pull Sync bit is sent, following the Push Field, the MCU listens on the MI–Bus for serial data bits sent back from the previously addressed MC33192 device. This portion of the communication sequence starts the "Pull Field Data" since it represents information pulled from the addressed MC33192 and received by the MCU.

The address selected MC33192 device sends data, in the form of status bits, back to the MCU reporting the devices condition. At the end of the Push Field the MCU outputs a Pull Sync bit which signals the start of the Pull Field. In the Pull Field are three bits (S2, S1 and S0) which report the status of the previously addressed MC33192 according to Figure 3.

Figure 3. Pull Field Status Bits

S2	S1	S0	Status	Comments
0	0	0	Not used	
0	0	1	Free	
0	1	0	No Back EMF	Drivers and/or coils failed
0	1	1	Free	
1	0	0	Normal/OK	
1	0	1	Thermal	Chip temperature > 150°C
1	1	0	Programming	PROM energized
1	1	1	Selection failed	Noise on MI–Bus, failed or disconnected module

The positive edge of the Pull Sync pulse (set by the MCU) causes all Push Field Data sent to the selected MC33192 to be stored in the output latch circuit in time with the strobe pulse. This means the data bits are emitted in real time synchronization with the MCU's machine cycle. The strobe pulse occurs only after the Push Field sequence is validated by the address selected device.

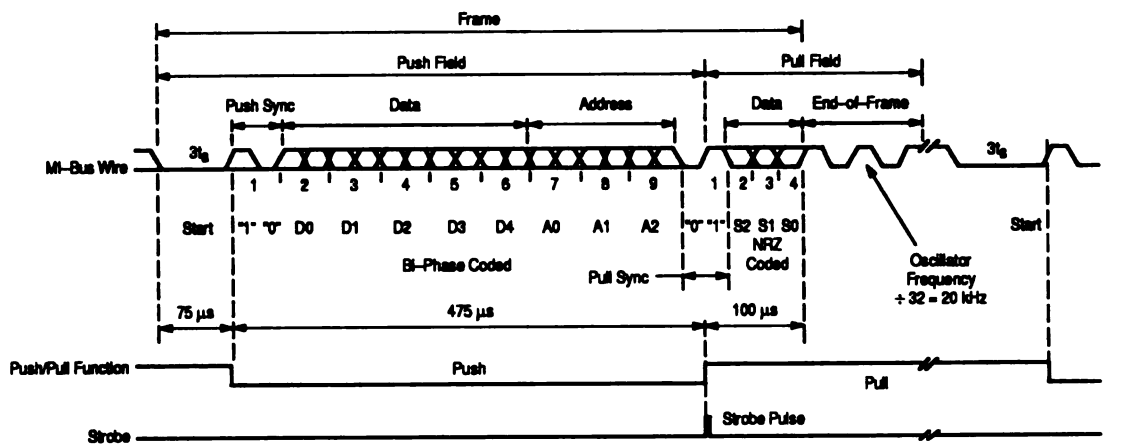
Message Validation

The communication between the MCU and the selected MC33192 device is valid only when the MCU reads (receives) the Pull Field Data having the correct codes (excluding the code "1–1–1" and "0–0–0") followed by an End–of–Frame signal. The frequency of the End–of–Frame signal may be a sub–multiple of the selected devices local oscillator or related to an internal or external analog parameter using a Voltage to Frequency Converter.

Error Detection

An error is detected when the Pull Field contains the code "1–1–1" followed by the End–of–Frame permanently tied to a logic "1" state (internally from 5.0 V through a pull–up resistor). This means the communication between the MCU and the selected device was not obtained.

Figure 4. MI–Bus Timing Diagram



There are four types of system error detections which are not mutually exclusive; These are:

1) Noise Detection

The system MC33192 slave devices receive the Push Field message from the MCU twice for each Time Slot (t_s) of the Bi–Phase Code. A receive error occurs when the two message samples fail to "logic wise" match. Noise and Bi–Phase detection are discussed further under Message Coding.

2) Bi–Phase Detection

The system slave devices receiving the Push Field message from the MCU detect the Bi–Phase Code. A detector error occurs when the two time slots of the Bi–Phase Code do not contain an Exclusive–OR logic function.

3) Field Check

A field error is detected when a fixed–form bit field contains an improper number of bits. A bit error can also be detected by the MCU during the Push Field. The MCU can simultaneously monitor the MI–Bus at the time it is sending data. A bit error is detected if the sent bit value does not match the value which was monitored.

4) Urgent Output Disable

If the MI–Bus becomes shorted to ground, the slave device outputs will be disabled after a period of $9t_s$. The MCU itself can take advantage of this feature to "globally" disable the outputs of all system slave devices by keeping the MI–Bus at a logic "0" level for a duration of $9t_s$ or more. Normal operation is resumed when the MCU sends a "standard" instruction over the MI–Bus.

Basic Stepper Motor Construction and Operation

Stepper motors are constructed with a permanent magnet rotor magnetized with the same number of pole pairs as contained in one stator coil section. Operationally, stepper motors rotate at constant incremental angles by stepping one step every time the current switches discretely in one stator field coil causing the North–South stator field to rotate either clockwise or counter–clockwise causing the permanent magnet rotor to follow (see Figure 5). For simplicity, assume the starting condition of the A1 to A2 stator field to be top to bottom polarized N to S and the B1 to B2 stator field to be left to right polarized N to S. The resulting stator field will produce a vector which points in the direction of position 3. The rotor will, in this case, be in the position shown in Figure 5 (pointing to position 1). This initial condition corresponds to that of step 1 in Figure 6. As the direction of current flow in the B1 to B2 stator field is reversed, the field polarity of the B1 to B2 also reverses and is left to right polarized S to N. This causes the resulting stator field vector to point in the direction of position 4. This in turn causes the N–S rotor to follow and rotate 90° in a clockwise direction and point in the direction of position 2. This condition corresponds to step 2 of Figure 6. Continued clockwise rotor steps will be experienced as the stator field continues to be incrementally rotated as shown in steps 3, 4, 5, etc. of Figure 6. The 90° steps in this simplistic example constitute "full steps". It is to be noticed that both coils, in the foregoing full step example, were simultaneously energized in one of two directions. It is possible to increment the rotor in 45° "intermediate steps" or "half steps" by alternately energizing only one stator coil at a time in the appropriate direction while turning the other stator coil off. The drive signals for Half Step operation are shown in Figure 7. The Power output stages of the MC33192 consist of two H–Bridges capable of driving two–phase bi–polar permanent magnet motors in either half or full step increment.

Figure 5. Permanent Magnet Stepper Motor

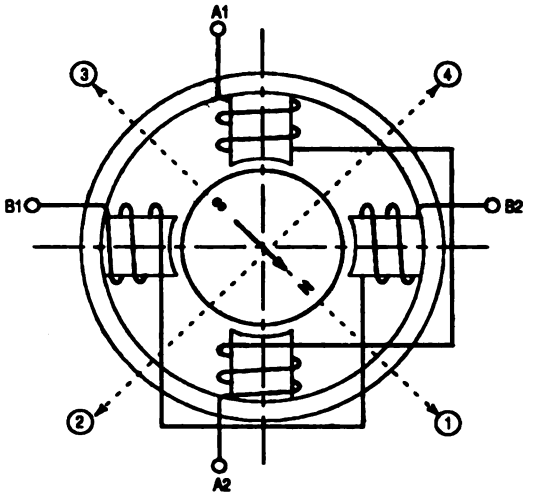


Figure 6. 4–Step "Full Step" Operation

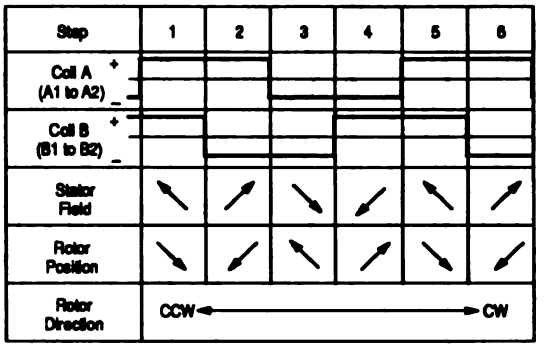
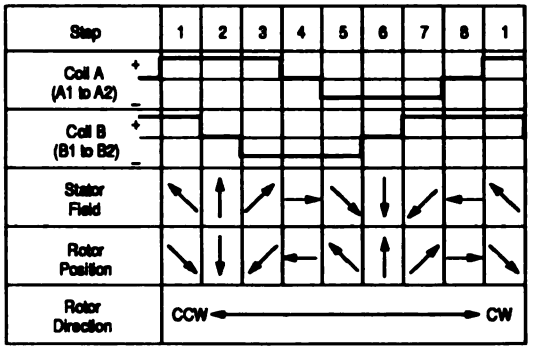


Figure 7. 8–Step "Half Step" Operation



Permanent magnetic stepping motors exhibit the characteristic ability to hold a shaft rotor position with or without a stator coil being energized. Normally the shaft holding ability of the motor with a stator coil energized is referred to as "Holding Torque" while "Residual Torque" or "Detent Torque" refers to the shaft holding ability when a stator coil is not energized. The Holding Torque value is dependent on the interactive magnetic force created by the resulting energized stator fields with that of the permanent magnet rotor. The Residual Torque is a function of the physical size and composition of the permanent magnet rotor material coupled with its intrinsic magnetic attraction for the un–energized stator core material and as a result, the weaker of the two torques.

It is to be noted when using half step operation, only one coil is energized during alternate step periods which produces a somewhat weaker Holding Torque. Holding Torque is maximized when both coils are simultaneously energized. In addition, since each winding and resulting flux conditions are not perfectly matched for each half step, incremental accuracy is not as good as when full stepping.

Two Phase Drive Signals

The DIR1 and DIR2 bits in the Data Frame of the Push Field determine the direction of H–Bridge current flow, and thus the magnetic field polarization of the stator coils, for H–Bridge outputs "A" and "B" respectively. The directional signals DIR1 and DIR2, generated by the MCU, communicate over the MI–Bus to control the two H–Bridge power output stages of the MC33192 to drive two phase bipolar permanent magnet motors. Figure 8 shows the MC33192 truth table to accomplish incremental stepping of the motor in a clockwise or counter–clockwise direction in either half or full step modes. The stator field polarization and rotor position are also shown for reference relative to the basic stepper motor of Figure 5.

Figure 8. Truth Table and Serial Push Field Data Bits For Sequential Stepping

		Push Field Bits											
Step		D0	D1	D2	D3	D4	H–Bridge Outputs				Stator Field (Note 2)	Rotor Position (Note 2)	Direction of Shaft Rotation
Full	Half	Inh1	DIR1	E	DIR2	Inh2	A1	A2	B1	B2			
1	1	1	0	1	0	1	1	0	1	0	↖	↘	CCW ↑
–	2	1	0	1	X	0	1	0	Z	Z	↑	↓	
2	3	1	0	1	1	1	1	0	0	1	↗	↙	
–	4	0	X	1	1	1	Z	Z	0	1	→	←	
3	5	1	1	1	1	1	0	1	0	1	↘	↖	
–	6	1	1	1	X	0	0	1	0	0	↓	↑	
4	7	1	1	1	0	1	0	1	1	0	↙	↗	↓ CW
–	8	0	X	1	0	1	Z	Z	1	0	←	→	
		0	X	X	X	0	Z	Z	Z	Z			
		1	1	0	1	1	Z	1	Z	1			
		1	0	0	0	1	1	Z	1	Z			
		1	1	0	0	0	Z	1	Z	Z			
		0	0	0	1	1	Z	Z	Z	1			

NOTES: 1. X = Don't care; Z = High impedance; 1 = High (active "on") state; 0 = Low (inactive "off") state.
2. The stator field direction and position of the rotor are shown for explanation purposes and relative to the basic stepper motor shown in Figure 3.
3. DIR1 establishes the direction of current flow in H–Bridge "A".
4. DIR2 establishes the direction of current flow in H–Bridge "B".

MI–Bus Interface Description

The MI–Bus interface shown in Figure 9 is made up of a single NPN transistor (Q1). The two main functions of this NPN transistor are:

1) To drive the MI–Bus during the Push Field with approximately 20 mA of current while also exhibiting low saturation characteristics ($V_{CE(sat)}$).

2) To protect the Input/Output (I/O) pin of the MCU against any Electro–Magnetic Interference (EMI) captured on the bus wire.

Without the NPN transistor, the MCU could be destroyed as a result of receiving excessive EMI energy present on the bus. In addition, the transistor blocks the MCU from receiving EMI signals which could erroneously change the data direction register of the MCU I/O.

The MCU input pin (P_{in}), used to read the Pull Field of the MI–Bus, is protected by two diodes (D2 and D3) and two resistors (R5 and R6). Any transient EMI generated voltage present on the bus is clamped by the two diodes to a windowed voltage value not to be greater than the V_{DD} or less than the V_{SS} supply voltages of the MCU.

MI–Bus Levels

The MI–Bus can have one of two valid logic states, recessive or dominant. The recessive state corresponds to a Logic "1" and is obtained through use of a 10 kΩ pull–up resistor (R9) to 5.0 V. The dominant state corresponds to a Logic "0" which represents a voltage less than 0.3 V and created by the $V_{CE(sat)}$ of Q1.

MI–Bus Overvoltage Protection

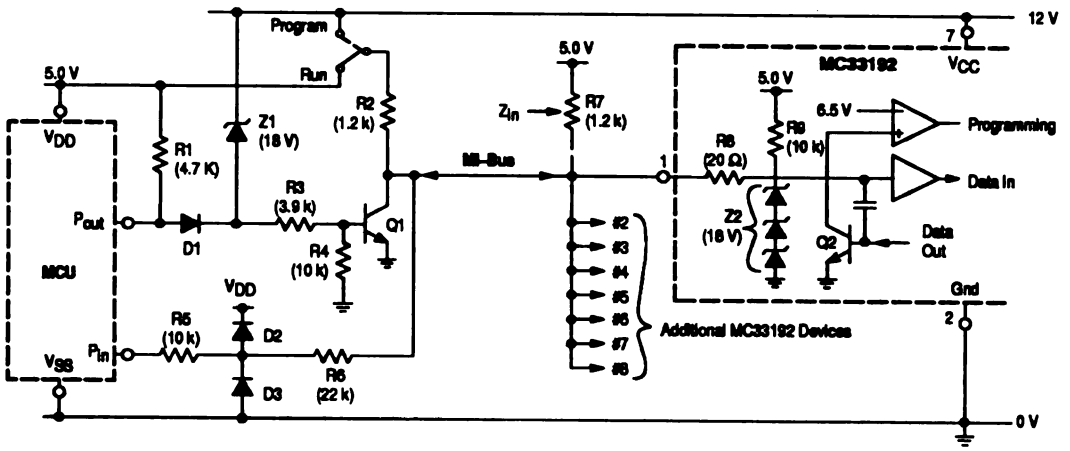
An external zener diode (Z1) is incorporated in the interface circuit so as to protect the MCU output pin (P_{out}) from overvoltages commonly encountered in automotive applications as a result of "Load Dump" and "Jump Start" conditions. Load Dump is defined as the inductive transient generated on the battery line as a result of opening the battery connection while the alternator system is producing charge current. Jump Start overvoltages are the result of paralleling the installed automotive battery, through the use of "jumper cables", to an external voltage source in excess of the vehicles nominal system voltage. For 12 V automotive systems, it is common for 24 V "jump start" voltages to be used.

When an overvoltage situation (>18 V) exists, due to a load dump or jump start condition, the zener diode (Z1) is activated and supplies base current to turn on the NPN transistor Q1 causing the bus to be pulled to less than 0.3 V producing a Logic "0" on the MI–Bus. After a duration corresponding to $8t_6$ (200 µs) of continuous Logic "0" on the bus all MC33192 devices will disable their outputs. Normal operation is resumed, following the overvoltage, by the MCU sending out a "standard" message instruction.

MI–Bus Termination Network

The MI–Bus is resistively loaded according to the number of MC33192 devices installed on the bus. Each MC33192 has an internal 10 kΩ pull–up resistor to 5.0 V. An external pull–up resistor (R7) is recommended to be used to optimally adjust termination of the bus for a load resistance of 600 Ω.

Figure 9. MI–Bus MCU Interface



MESSAGE CODING

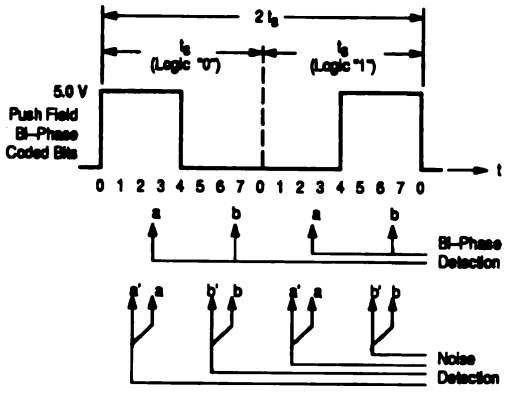
Bi–Phase Coding and Detection

The Manchester Bi–Phase code shown in Figure 10 requires two time slots ($2t_s$) to encode a single data bit. This allows detection of a single error at the time slot level. The logic levels "1" or "0" are determined by the organization of the two time slots. These always have complementary logic levels of either zero volts or plus five volts, which are detected using an Exclusive OR detection circuit during the Push Field sequence. A "1" bit is detected when the first time slot is set to a zero logic state (0 V) followed by the second time slot set to a logic state one (5.0 V). Conversely, a "0" bit is detected when the first time slot is set to the logic state "one" (5.0 V) followed by a second time slot set to a "zero" logic state (0 V). For these two bits are Exclusive–ORs of each other.

The addressed devices receiving the Push Field detect the Bi–Phase code. Bi–Phase detection involves the sampling of the Push Field Bi–Phase code twice (a and b) for each time slot. A code error occurs when the two time slots of the Bi–Phase do not follow a logical Exclusive–OR function (see Figure 10).

Noise monitoring is accomplished by sampling the Push Field Bi–Phase code twice (a and a') and (b and b') during each time slot. A noise error is detected if the two sample values do not have the same logical level.

Figure 10. Noise/Bi–Phase Detection



Each message frame consists of two fields: The Push Field, in which data and addresses are transferred by the MCU to the slave device; and the Pull Field, in which serial data is transferred back to the MCU from the address selected slave device. The message frame is broken down into seven individual field segments as indicated in Figure 4 (Start, Push Field Sync, Push Field Data, Push Field Address, Pull Field Sync, Pull Field Data, and End–of–Frame). The following lists the bit size and function of each of these segments:

1) Start is the start of message and consists of three time slots ($3t_s$) having the dominant Logic "0" state of less than 0.3 V. Holding the MI–Bus at ground for three time slots ($3t_s$) marks the beginning of the message frame by violating the law of the Manchester Code.

2) Push Field Sync is a single bit which establishes initial timing for the Push Field Data to follow.

3) Push Field Data is comprised of five serial data bit fields (D0, D1, D2, D3 and D4) which comprise the instruction set defining the configuration and condition of the two H–Bridge output stages.

4) Push Field Address is comprised of three serial data bit fields (A0, A1 and A2) which define the address or name of a MC33192 on the MI–Bus.

5) Pull Field Sync is a single bit which establishes the end of the Push Field and the initial start timing for the Pull Field Data to follow.

6) Pull Field Data is made up of three serial data bit fields (S2, S1 and S0) which contain the existing status information of an addressed MC33192.

7) End–of–Frame field is a signal which communicates to the MCU that the status information sent by the MC33192 is complete.

The Push Field Sync bit, Push Field Data bits, Push Field Address bits, Pull Field Sync bit are all coded by the Manchester Bi–Phase L Code. The Pull Field Data bits are Non–Return to Zero (NRZ) coded. The End–of Frame field is a square wave signal with a frequency of 20 kHz or higher so as to avoid a condition which causes a bus violation.

The Manchester Bi–Phase L code requires two time slots ($2t_s$) to encode a single bit. This allows a single error to be detected during the time slot.

Address Programming involves the use of three instructions. Refer to Figure10.

First Instruction Set the MI–Bus continuously at 12 V. This places the MC33192 in the programming mode. Programming is possible only when the MI–Bus is at 12 V.

Next, the MCU serially enters "Logic Zeros" in all five Push Field Data bit positions (D0, D1, D2, D3 and D4) followed by the designated address value in the Push Field Address positions (A0, A1, & A2).

The MCU now waits 275 µs before starting the second instruction. The total of the Pull time, Delay time, and Bus Violation time (V) of the second instruction (150 µs, 275 µs and 75 µs respectively) will cause the memory cell to be energized for 500 µs. During the first 150 µs of this time, the MCU is checking the Pull Field Data Bits S2, S1 and S0 looking for the programming code "110" to indicate complete activation of the memory cell.

Second Instruction (MI–Bus voltage remaining at 12 V)

The MCU repeats the same Push Field instruction as previously sent in the First Instruction; entering all "Logic Zeros" in the Push Field Data positions followed by the designated Push Field Address value in the address positions.

Again, the MCU waits for the Pull, Delay, and Bus violation time while checking the Pull Field Data bits looking for the programming code "110" code. The MCU must repeat the initial Push Field Address instruction until a "110" code is received before advancing to the Third Instruction.

Third Instruction The MI–Bus voltage is lowered to 5.0 V.

The MCU serially loads "Logic Zeros" in all five Push Field Data bit positions followed by the programmed address in the Push Field Address positions. The MCU then checks the Pull Field Address status bits looking this time for the

programming OK code "100" indicating the address programming to be executed.

The First and Second Instructions must be repeated until the MCU successfully receives the programming code "100". Address programming is not complete until a "100" OK status is received by the MCU with the MI–Bus voltage at 5.0 V.

Overwrite–Bit Programming involves the use of two instructions. See Figure 11.

First Instruction Have the MI–Bus continuously set at 12 V so as to have the MC33192 in the programming mode. Programming can only be accomplished with the MI–Bus at 12 V.

The MCU serially enters "Logic Zeros" for the Push Field Data bits D0, D1, D2 and D3 and a Logic "1" for D4 bit followed by the programmed address bits A0, A1 and A2.

The MCU now waits 275 µs before starting the second instruction. The total of the Pull time, Delay time, and Bus Violation time (V) of the second instruction (150 µs, 275 µs and 75 µs respectively) will cause the memory cell to be energized for 500 µs. During the first 150 µs of this time, the MCU is checking the Pull Field Data Bits for the status of bits S2, S1 and S0 looking for the **programming code "110"** to indicate complete activation of the memory cell.

Second Instruction (MI–Bus remaining at 12 V)

The MCU repeats the first instruction outlined above until the **programming OK code "100"** is sent back to the MCU from the selected MC33192 indicating the overwrite–bit protection to be programmed. If after eight repeat instructions, the programming code "110" or the OK code "100" is not generated four times in succession, programming of the MC33192 has failed. If this occurs, the Overwrite–Bit Programming sequence should be reviewed and re–started from the beginning.

H–Bridge Output

The H–Bridge output drive circuit and associated diagnostic encoder are shown in Figure 12. The H–Bridge output uses internal diode clamps (D1, D2, D3, D4) to provide transient protection of the output transistors necessary when switching inductive loads associated with stepper motors.

Back EMF Detection

Three different Back EMF currents can occur depending on whether the motor is running or manner in which it is being stopped. Referring to Figure 12; When the Dir1 bit is set to logic 0, the direction of current flow will be from V_{CC} through transistor Q2, Coil A (A1 to A2), and transistor Q4 to ground.

1) **Fast Decay** (when transistors Q1, Q2, Q3 and Q4 are switched off).

When the current flowing in the coil is stopped by setting the Inh1 bit to logic 0, the back EMF current will circulate through the voltage supply (V_{CC}) and diodes D1 and D3. At that time, the voltage developed across the diode D1 is detected by transistor Q6. The generated voltage pulse of Q6 is then encoded and sent, in the Pull–Field, to the microprocessor.

2) **Slow Decay** (Q3 and Q4 are switched off)

When the current flowing in the coil is stopped by setting the E bit to logic 0, the back EMF current will circulate through the diode D1 and transistor Q2 which is already switched on.

3) **When Motor is Running**

The rotational direction of the motor changes whenever the Dir bit state is changed. When the Dir bit is changed from a logic 0 to a logic 1, transistors Q2 and Q4 are switched off and transistors Q1 and Q3 are switched on. At this time, the back EMF current will circulate from ground through diodes D1 and D3 to the voltage supply (V_{CC}). In all cases, the back EMF currents will be detected by transistors Q5 and Q6.

Figure 11. Address Programming Diagram

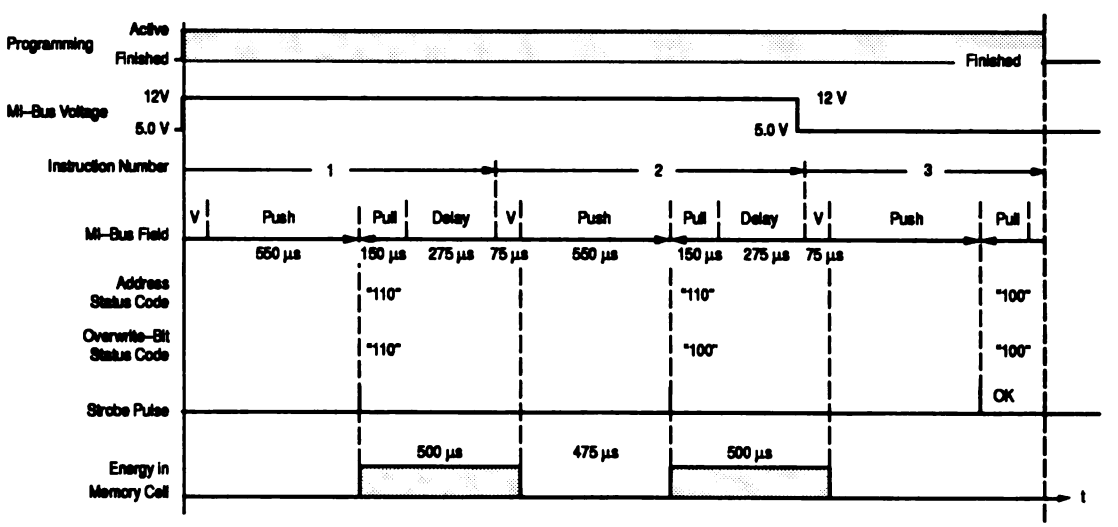
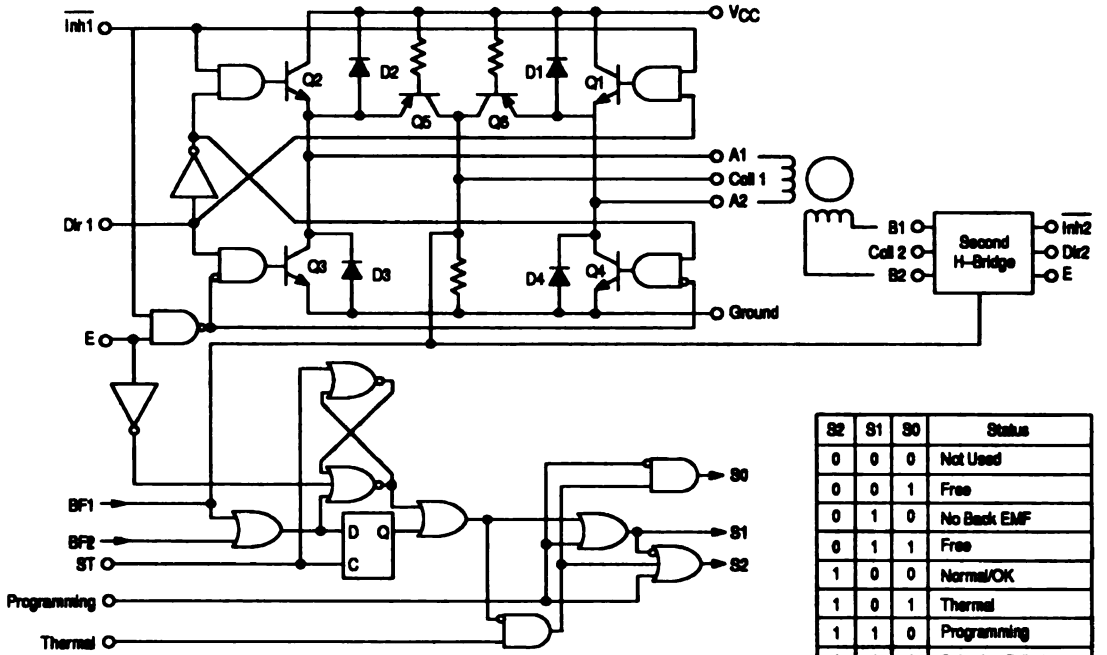
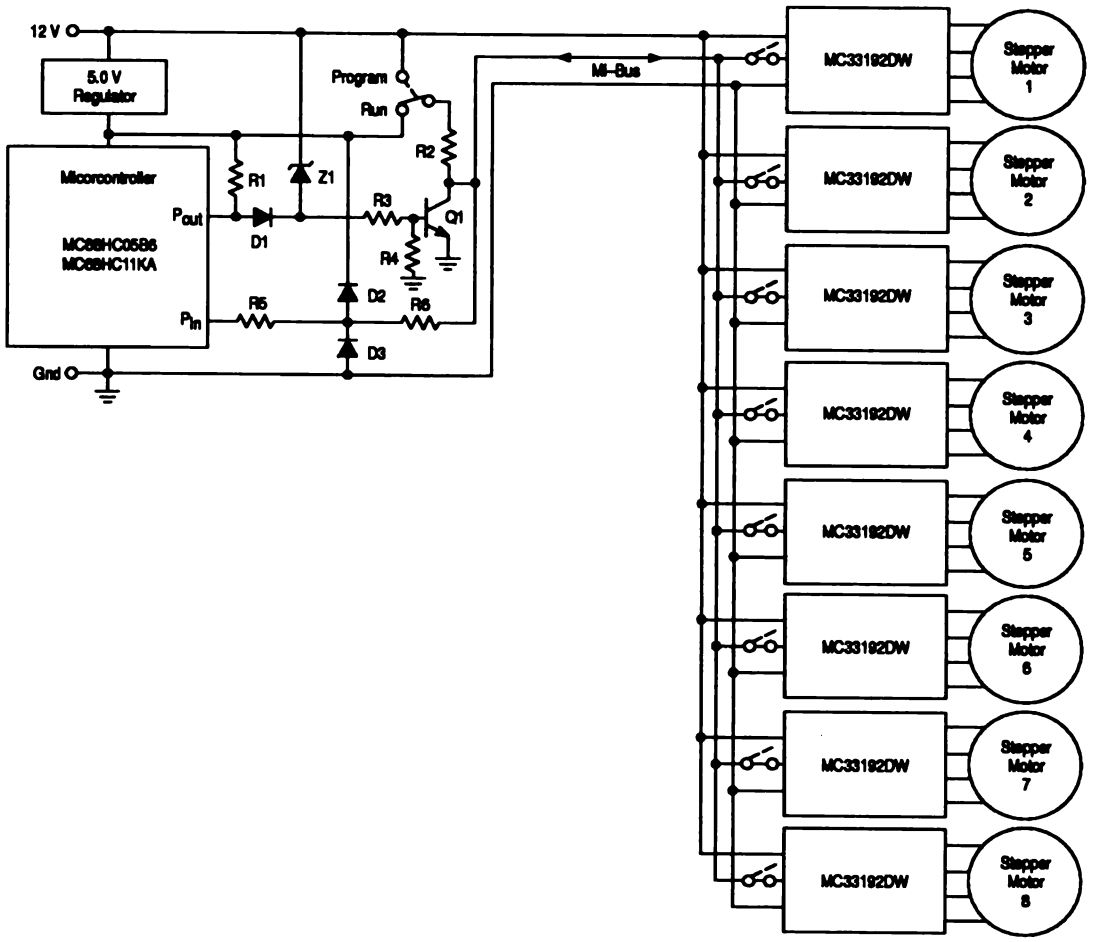


Figure 12. H–Bridge Output Drive Circuit and Diagnostic Encoder



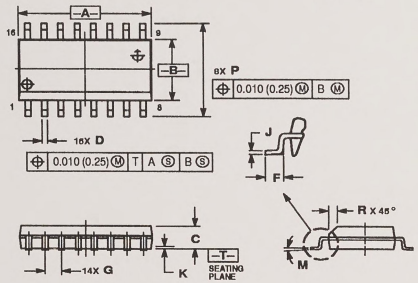
S2	S1	S0	Status
0	0	0	Not Used
0	0	1	Free
0	1	0	No Back EMF
0	1	1	Free
1	0	0	Normal/OK
1	0	1	Thermal
1	1	0	Programming
1	1	1	Selection Failed

Figure 13. Single Wire MI–Bus Control of 8 Stepper Motors



OUTLINE DIMENSIONS

DW SUFFIX
PLASTIC PACKAGE
CASE 751G–02
(SO–16L)
ISSUE A



NOTES:
1. DIMENSIONING AND TOLERANCING PER ANSI Y14.5M, 1982.
2. CONTROLLING DIMENSION: MILLIMETER.
3. DIMENSIONS A AND B DO NOT INCLUDE MOLD PROTRUSION.
4. MAXIMUM MOLD PROTRUSION 0.15 (0.006) PER SIDE.
5. DIMENSION D DOES NOT INCLUDE DAMBAR PROTRUSION. ALLOWABLE DAMBAR PROTRUSION SHALL BE 0.13 (0.005) TOTAL IN EXCESS OF D DIMENSION AT MAXIMUM MATERIAL CONDITION.

	MILLIMETERS		INCHES	
DIM	MIN	MAX	MIN	MAX
A	10.15	10.45	0.400	0.411
B	7.40	7.60	0.292	0.299
C	2.35	2.65	0.093	0.104
D	0.35	0.49	0.014	0.019
F	0.50	0.90	0.020	0.035
G	1.27 BSC		0.050 BSC	
J	0.25	0.32	0.010	0.012
K	0.10	0.25	0.004	0.009
M	0 °	7 °	0 °	7 °
P	10.05	10.55	0.395	0.415
R	0.25	0.75	0.010	0.029

How to reach us:
USA/EUROPE/Locations Not Listed: Motorola Literature Distribution;
P.O. Box 20912; Phoenix, Arizona 85036. 1–800–441–2447 or 602–303–5454

MFAX: RMFAX0@email.sps.mot.com – TOUCHTONE 602–244–6609
INTERNET: http://Design–NET.com

JAPAN: Nippon Motorola Ltd.; Tatsumi–SPD–JLDC, 6F Seibu–Butsuryu–Center, 3–14–2 Tatsumi Koto–Ku, Tokyo 135, Japan. 03–81–3521–8315

ASIA/PACIFIC: Motorola Semiconductors H.K. Ltd.; 8B Tai Ping Industrial Park, 51 Ting Kok Road, Tai Po, N.T., Hong Kong. 852–26629298

MOTOROLA

◊

M33192/D

BIBLIOGRAPHY

Bibliography

[1] Philip Koopman, "Embedded System Design Issues(the Rest of the Story)",*International Conference on Computer Design*, IEEE Computer Society, Los Alamitos CA, 1996, pp. 310-317.

[2] "POLIS: a framework for hardware/software co-design of embedded systems." [Online] Available http://www-cad.eecs.berkeley.edu/Respep/Research/hsc/polis_files.html.

[3] "Hardware-Software Co-Design Study." [Online] Available http://www.mcc.com/projects/hwsw-codesign/std_prop.html.

[4] POLIS User's Manual. [Online] Available http://www-cad.eecs.berkeley.edu/Respep/Research/hsc/polis _files.html.

[5] Felice Balarin, Massimiliano Chiodo, et al, *"Hardware-Software Co-design Of Embedded Systems, The POLIS Approach"*, Kluwer Academic Publishers, 1997.

[6] D. Harel, A. Naamad, "The STATEMATE semantics of statecharts." [Online] Available http://www.acm.org/pubs/toc/Abstracts/tosem/235322.html

[7] VIS(Verification Interacting with Synthesis). [Online] Available"http://www-cad.eecs.berkeley.edu/Respep/Research/vis/index.html

[8] "The ESTEREL Language.' [Online] Available http://www.inria.fr/meije/esterel/

[9] C. Kuttner, "Hardware-Software Codesign Using Processor Synthesis", *IEEE Design & Test of Computers*, Fall 1996, pp. 43-53.

[10] "A D&T Roundtable, Hardware-Software Codesign", *IEEE Design & Test of Computers*, January-March 1997, 75-83.

[11] "ASICs and Design Tools, Design Report, '97 Paris Forum ", *Computer Design*, June 1997, pp. 53-64.

[12] C.A.R. Hoare, "Hardware and Software : Closing the gap", *Transputer communications*, June 1994, pp. 69-90.

[13] S. Schulz, J. Rosenblit, et al, "Model-Based Codesign", *Computer*, August 1998, pp. 60-67.

[14] K. Buchenrieder, C. Veith, "A Prototyping Environment for Control-Oriented HW/SW Systems", ACM 1994, pp. 60-65.

[15] "EDA Watch, Long Overdue Unified HW/SW Co-Design Language Comes to Light", *Electronic Design*, May 13, 1998, pp. 60-62.

[16] D. Gajski, F.Vahid, "Specification and Design of Embedded Hardware Software Systems", *IEEE Design & Test of Computers*, Spring 1995, pp. 53-67.

[17] D.Gajski, F. Vahid, et al, *Specification and Design of Embedded Systems*, Prentice-Hall, Englewood Cliffs, N.J.,1994.

[18] M. Srivatsava, R. Brodersen, "Rapid Prototyping of Hardware and Software in a Unified Framework", *Proc. Int'l Conf. Computer-Aided Design*, IEEE CS Press, 1992, pp 152-155.

[19] D. Thomas, J. Adams, et al, "A Model and Methodology for Hardware/Software Codesign", *IEEE Design & Test of Computers*, Vol. 10, No. 3, Sept. 1993, pp. 6-15.

[20] R.Ernst, J. Henkel, et al, "Hardware-Software Cosynthesis for Micro-Controllers," *IEEE Design & Test of Computers*, Vol. 10, No. 4, Dec. 1993, pp. 64-75.

[21] R.Gupta, D. Micheli, "Hardware-Software Cosynthesis for Digital Systems," *IEEE Design & Test of Computers*, Vol. 10, No. 3, Oct. 1993, pp. 29-41..

[22] K. Buchenreider and C. Veith, "CODES: A Practical Concurrent Design Environment,"

[23] S.L. Coumeri, D.E. Thomas, "A Simulation Environment for Hardware-Software Codesign",*Euro DAC*, IEEE Computer Society, Los Alamitos CA, 1995, pp. 58-63.

[24] "HW/SW Co-Design for Embedded Systems", Alberto S-V, ILP, March 1995.

[25] "Ptolemy". [Online] Available http://ptolemy.eecs.berkeley.edu

[26] "The Almagest: A Manual for Ptolemy". [Online] Available http://ptolemy.eecs.berkeley.edu/papers/almagest/docs/user

[27] "The Esterel v5 Language Primer". [Online] Available http://www.inria.fr/meije/esterel

[28] [Online]Available http://www.eagledes.com

[29] F. Balarin, H. Hsieh, et al, "Formal verification of embedded systems based on CFSM networks," *Proceedings of the Design Automation Conference,* 1996.

[30] M. Chiodo, P. Giusto, et al, "A formal specification model for hardware/software codesign for embedded systems," *IEEE Micro,* 14(4):26-36, August 1994.

[31] *Motorola Analog IC Device Data,* MC33192, Motorola Inc., 1996.

[32] Michel Burri and Dr. Pascal Renard, "Single wire MI Bus controlling stepper motors", *Motorola Semiconductor Application Note, AN475,* Motorola Ltd., 1993.

[33] "The Handy Board". [Online] Available http://lcs.www.media.mit.edu/groups/el/Projects/handy-board

[34] "Interactive C for the Handy Board Manual". [Online] Available http://lcs.www.media.mit.edu/groups/el/Projects /handy-board/techdocs/index.html