ISSN 0105-8517

FIRMWARE DEVELOPMENT SYSTEMS

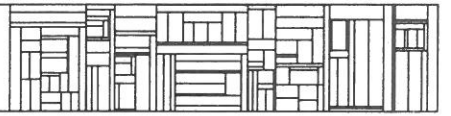
A Survey

by

Peter Kornerup

DAIMI PB-127
November 1980

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55



Firmware Development Systems, a Survey

Peter Kornerup

Abstract:

Selected firmware development systems are investigated, with special emphasis on the environment in which the firmware is existing during testing and final installation. In particular the RIKKE/MATHILDA system is described. Development systems for user microprogrammable processors, as well as for dedicated control systems (custom-designed) are surveyed and classified. Particular problem areas are illustrated and discussed.

1. Introduction

Although microprogramming as a technology for the implementation of the control of a computer has existed for almost as long as the electronic computer itself, the tools available for development, debugging and verification of firmware have not in general reached the state of equivalent tools available for software.

The firmware development was for a long time intimately connected with the design of the processor, as was the debugging of the microprograms with the debugging of the processor hardware. With diode matrices, or most other discrete read-only memory technology for storing the microprogram, it does not seem reasonable to debug the host processor by special microprograms, before the microprogram realizing the target architecture is "loaded", in particular since the latter anyway was going to be the only "program" ever to run on the host.

With exchangeable or re-writeable control stores the situation is quite different. The interest in emulating other target architectures than the original target machine, or in general to support a variety of applications on the same host, creates a need for development and debugging tools.

A firmware development system may consist of tools for programming, loading, testing (simulation/debugging), verification and installation of firmware.

The history of firmware development tools seems to be parallel to that of ordinary program construction tools. It is interesting to notice, that although most early microprogrammable hosts had horrible architectures, complicated instruction sets and tricky timing, how primitive tools were used, and how slowly the tools have evolved. One reason for this may be the fact that most pieces of firmware are of a very limited size, and of a very simple structure. The simple and modular structure of an interpreter in general makes the construction of such a piece of firmware a reasonably comprehensible task.

The most essential tool for firmware development is some sort of translator, assembler or compiler, which to a varying degree can relieve the programmer of the burden of the detailed task of expressing the program in terms of field encodings of the primitive operations. Besides introducing symbolic opcodes and labelling, microassemblers most often furthermore are used to "hide away" certain peculiarities of the underlying architecture. E.g. assemblers were often used to handle the ackward addressing and instruction allocation caused by the

early technique of conditional sequencing, where a jump address is formed by concatenation of litteral fields from the microinstructions, with flagbits from processor status. In this paper we will not discuss the pros and cons of high level languages for microprograming. Although the languages available do play a crucial role in the process of firmware development, we will try to concentrate on the environment where the firmware is going to exist during its development and after its completion.

If there is an area in which firmware engineering is different from software engineering, it is to be found in the environment in which programs are tested and installed. In systems which allow user microprogramming it is intentionally such that the existence of a piece of firmware, and its invocation, is supposed to be transparent to the ordinary programmer of the (software) system. In a multi-interpreter system it is the task of the system to load and invoke the appropriate pieces of firmware, whenever the user initiates the execution of a piece of software. The fact that the operating system executes by means of one (firmware) interpreter, a particular compiler possibly runs on another, and the code generated is intended to execute on a third, has to be transparent to the user who calls upon the system to compile and execute his source program.

The protection and integrity of ordinary user programs are secured through the system, which also provides an environment of facilities to call upon. This environment has been created through interpretation on top of the hardware, which allows the system to monitor and supervise the user program. In contrast, the microprogram has to execute in an environment which provides little or no protection, where in general the available support is very limited, and communication with the "user" (i.e. the firmware-programmer) is either restricted to lights and switches or may be severely hampered by the layers of system above the firmware (unless of course specific tools are available, e.g. a firmware debugger). Also I/O in general may be complicated, either because it has to be handled at a very low level, or because it has to go through the ordinary system I/O routines (i.e. calls have to "go upwards").

With the advent of bit-slice technology, history has repeated itself. This technology may be used to architecture "ordinary" microprogrammable processors, i.e. intended to realize a target architecture through interpretation. However, microprogrammed bit-sliced control units may be used directly in various controllers, requiring complicated sequencing. For such purposes the "firmware level" becomes "the applications level", and the firmware host will in general be designed more or less from scratch for each application. This is almost the same situation as the one faced in the desing of the early microprogrammed processors, except for the fact that the "building blocks" of

the hardware design today are LSI circuits, whose level of functionality assures more homogeneity across designs.

In either of these situations, the firmware of such control units will most often be fixed in ROM storage. Design, development, testing and verification of the firmware will have to take place in some way or other external to the final system, where the firmware is going to reside. Since the firmware host architecture changes from one application to the next, support systems will have to be parameterizable to amortize their cost, and reduce the time needed to create support for the individual design situation.

In the rest of this paper we will discuss firmware development and test systems, based on actual systems. First we will treat user-microprogrammable systems, where apparently most progress has been made. One particular sytem, the RIKKE-MATHILDA system at Aarhus will be described to some depth, as it has not been reported on before. A sample of other development systems will then be surveyed, to illustrate complementary facilities.

Systems to aid in firmware development for "custom-designed" hosts, in particular ROM based controller applications, will be summarized in short. The tools reported on in this area seem so far only to be quite limited in scope and complexity, but we will try to see what can be learned from the systems available for user microprogrammable systems.

2. The RIKKE-MATHILDA System

The microprogramming lab facilities at Aarhus consist of two (almost identical) microprogrammable processors RIKKE and MATHILDA, which are intimately connected, sharing one memory system WS (Wide Store). Both processors have the same architectural structure, but differ in the bit-width of their datapaths and registers. MATHILDA has a width of 64 bits, and can be considered a powerful functional unit of RIKKE (usually M runs as a slave of R), and has as such no I/O devices connected. RIKKE has a 16 bit datapath and 16 bit wide registers. Attached to RIKKE is a private storage system MS (32K, 16 bit), a variety of standard I/O devices (CRT, lineprinter, papertape I/O and disk controller), and a high-speed file transmission link to the departmental DEC10 system. The common storage WS has 4 ports, each of which has a block-transfer capability. It is organized as two 16K banks of width 64 bits, however it is possible to write selectively into any combination of 16 bit fields of a storage word, leaving the contents of the remaining fields unchanged. RIKKE

and MATHILDA may hence view WS as 128K-16 bit or 32K-64 bit respectively, and may communicate through WS, as well as through direct 16 bit connections (with busy/done flags). The disk-system is also connected to one of the WS ports, acting as a data-channel, but transfers can only be initiated from RIKKE.

The architecture of RIKKE and MATHILDA [10] is organized around a single datapath connecting storage and functional resources, including two 256 word local storages (scratchpad registers), registers with shift capability, and an ALU where one operand comes from a 4 word storage. The datapath itself contains a barrel shifter and various masking facilities. Control of the resources is either immediate (from microinstruction) or residual, and in the latter case originating from various sources, including small dedicated 16 word "register groups".

Control store is 64 bit wide (2K in RIKKE, 1K in MATHILDA), and permits pre-fetch of instructions. The processor operates with a polyphase clock and fixed cycle length. The execution of the instruction can be divided in four steps: datapath transfer, clock1 microoperations (mops), clock2 mops and sequencing. Due to the amount of parallelism (up to 6 mops may be specified in one instruction), the class-division of mops, and in particular the possibilities for residual control, makes very tight coding possible (and often used). With proper "set-up" many loops used in coding of arithmetic instructions (e.g. multiplication and division) are "one-liners", i.e. instructions that loop on themselves. Residual control is also useful in the decoding of instructions, but in general it complicates the firmware development process, and the environment in which the firmware is to exist. Sequencing is quite powerful, and includes the possibility of self-relative addressing, indexing (case-branch) and two 16 level return jump stacks. The "state-vector" of MATHILDA contains approx. 8000 bits, not counting the two local stores (32K bits) and control store (64K bits). Hence the only feasible way to realize a context switch is to have each piece of firmware establish its own environment "on top of" the standard environment used by all firmware (and in particular the I/O nucleus in RIKKE [12]). There are no hardware facilities to protect any piece of firmware and its resources from a "bad" piece of firmware, not even control store is protected.

The hardware has evolved over a long period, starting with the construction of RIKKE in 1972-73, as a prototype of the MATHILDA design. With some design changes MATHILDA was constructed and became available early 76, together with WS. Later the disk system and the block-transfer capability on WS has been added, just as a number of minor updates on RIKKE and additions to MATHILDA have been made.

It was at an early stage decided to use the language BCPL [16] as the vehicle for systems programming on RIKKE, and a firmware interpreter for O-code (an intermediate language for BCPL) was implemented on RIKKE late 1973 [24]. Supported by the I/O Nucleus [12], the RIKKE OS was developed (based on the Oxford OS [21]) into a versatile single user operating system with a sophisticated file system. A variety of other firmware interpreters have been developed for experimental purposes, including lately a "twin-interpreter" supporting pure concurrency in Concurrent Pascal, running on the combination RIKKE-MATHILDA-WS. A Pascal-supporting interpreter on RIKKE, utilizing MATHILDA as a functional unit for non-standard arithmetic [11], is about to be completed. Experiments with migration of operating system functions have been performed in connection with the disk-drivers [23], with primitives for a relational database system [2], and a string search operator in the editor.

Except for some very first hardware test programs, all firmware for RIKKE and MATHILDA has been developed using the microassembler MARIA (existing in two versions) and in most cases debugged on the (batch) simulators for RIKKE and MATHILDA. Assemblers and simulators are implemented in BCPL, originally in 1973 on a CDC 6400, but later transferred to the departmental DEC10 System [22], as well as to a number of other installations to be used as a teaching and research tool in other departments.

The MARIA microassembler provides the same sort of facilities that are usually found in ordinary assemblers, but does not contain macrofacilities. Datapath transfer is specified as an assignment with a possible shift specification. The list of mops is free format, and their ordering is immaterial. Since some mops may be represented in any one of two or three fields, the assembler also has the task of "shuffling" around with such mops to make the whole list fit. Mops which specify load-actions with a source selection are specified as assignments, which are then represented as a combination of the load action and a setting of the proper source selection bits of the instruction. Also if the source is specified as the current microinstruction, i.e. as a literal, that constant is assembled into the proper field (and that field is marked as non-executable). The third part of the microinstruction specifies the sequencing part in the form of an if-then-else construction, with suitable defaults if not "all the power" is needed. The hardware provides the conditional selection among two address-computations (rather than addresses), whose binary specification is quite tricky and sometimes interrelated. A task of the assembler is to cover up such details, and, as in general, to detect resource conflicts.

A particular feature which the MARIA assembler provides is the possibility of environment initialization, which is not only available for initialization of

simulator resources (e.g. registers), but also provides initialization of resources in the physical processors. This is realized by supplying the source with "value" pseudoinstructions, which causes the assembler to append to the load file additional "initialization records", one for each resource whose initialization has been specified. Each record consists of an identification of the resource, and the necessary data. For each such record, the systems loads a small microprogram (usually only a few instructions) and supplies it with the data to initialize the resource (or part of it). Thus the user simply specifies how he wants the environment to look when his microprogram is initiated, and does not have to worry about doing it. And it is specified exactly the same way whether the microprogram source is to be assembled for the simulator or for the physical machine.

The simulators perform faithful simulations of the processors, but only crude simulations of I/O, since they do not attempt to simulate timings of the devices. The simulators operate in "batch mode", and devices are simulated by files. The RIKKE simulator only provides simulation of some of the early physical devices, as most user microprograms utilize the I/O nucleus. As the only "device" with which MATHILDA can communicate is RIKKE, corresponding files are provided, however there is no possibility to run the simulators of RIKKE and MATHILDA in parallel. The simulators provide a tracing of the user microprogram program counter as the default. All other output from simulators has to be specified in the source program to the assembler by means of "display" pseudo-instructions. The range of a display statement is the (static) interval from its occurrence in the source, until the next one is encountered. The display pseudo-instruction does not immediately specify what is to be dumped on output, it only references a "table" pseudo-instruction. In this way it is fairly easy to insert and delete the display statements in appropriate places. The table definitions, containing the details of which resources are to be dumped, then have to appear only once, and can be collected in one place.

An interactive simulator for the MATHILDA processor has been implemented on a Multics system but has so far not been transported to Aarhus, as the implementation heavily relies on the Multics file system (segments).

As the assembler/simulator system is available on the DEC10 System, microprograms are developed in a standard time sharing environment, and are normally tested on the appropriate simulator before the assembled code is transferred (through the transmission link) to the file system on RIKKE. Initially files were "transmitted" on paper tape, but it has never been found necessary to move the assembler to the RIKKE system, as it is physically located next door to the DEC10 system.

With the two interconnected microprogrammable processors available, basically three different modes of operation are at the programmer's disposal. He can implement a piece of firmware to run on RIKKE, under the support of the OS, but will to a large extent have to obey to the rules of the system. Or he can choose to implement his firmware on MATHILDA, in which case he can have an almost "naked" machine, but still with powerful support available from RIKKE. Finally both machines can be integrated in one system, either as two truly parallel processors, or MATHILDA as a functional unit of RIKKE.

The philosophy of the R/M system is to let the firmware programmer construct in BCPL (the system language) the environment in which the firmware is to exist. A few standard routines available allows the user on RIKKE to load and execute microcode in RIKKE, as well as in MATHILDA, and to communicate with the microcode.

The primitives for the control of microcode in RIKKE are provided by the following three BCPL functions:

```
SetupRikkeCS
LoadRikkeCS
CallRikkeCS
```

the first of which just initializes a table, used by the control store allocation (recall that it is a single-user system). LoadRikkeCS takes a filename as parameter (assumed to be a file containing assembled microcode) and performs the specified resource initializations and loading of microcode into control store. It returns (as the value of the function) entrypoint information if the call was successful, otherwise an error condition.

When microcode has been successfully loaded into RIKKE control store, it can be repeatedly invoked by calls of the form:

```
res := CallRikkeCS (entry,p),
```

where entry is a variable initialized by LoadRikkeCS to contain the entrypoint. The parameter p is supposed to be the name of an array (a BCPL vector) which is the communication area (in WS) between the BCPL level and micro level. When the microcode gets control through this call, the address of this area is found in a particular register (LR), and the contents of the two first locations have been transferred to two other registers (DS and VS). Upon completion of the user microcode, it just executes a particular "return from subroutine" jump, and whatever is in the register DS is delivered as the value of the CallRikkeCS

function call to the BCPL level. Of course, nothing prevents the microcode from accessing other parts of the communications vector, nor to interpret it in any way it wants, e.g. to use its contents as pointers to other data. The transfer to and from registers is just provided to facilitate the implementation of firmware routines with a minimal need for parameter transfer.

These routines are straightforward to use whenever a particular software level function is to be migrated into firmware. Notice that the communications vector is inside the address space of the BCPL program (which is at most 64K, 16 bit or 16K 64 bit of WS), but the firmware routine called upon may access WS beyond this space, or it may access MS (the 16 bit storage accessible only by RIKKE).

If the firmware called upon is an interpreter for another virtual machine, its address space may be disjoint from that of the BCPL program. The interpreter and the BCPL program may communicate as coroutines, since the interpreter may call back and request a service from the BCPL level system, in the following way. The interpreter returns to the BCPL level with a "result" (contents of DS), which specifies what service is wanted (and possibly parameters in the communications vector) and after the BCPL level has performed the requested action, it calls upon the interpreter again (possibly returning results of the action in the communication area).

The routines provided for loading and calling upon firmware in MATHILDA are basically equivalent, except for the differences caused by the asynchronous operation of the caller and callee:

SetupMatCS
LoadMatCS
CallMatCS
MatIdle
InMat64

The setup and loading is completely equivalent to the RIKKE routines, but CallMatCS differs in that it only invokes MATHILDA (assumed idle) and then immediately returns to the BCPL system (i.e. it does not wait for a result), allowing RIKKE to continue operation. The communications area consists of 64 bit words, i.e. they have to be handled by the BCPL system as four 16 bit words. The MatIdle function allows testing MATHILDA for completion, in which case InMat64 can read the contents of DS, and the communications area may be interpreted safely by the BCPL system. Notice that since operation of RIKKE and MATHILDA is asynchronous, any communication through the

shared memory WS has to be carefully programmed to assure mutual exclusion of access to shared data.

A set of extra facilities are provided for the debugging of firmware in MATHILDA, in the form of some "snooping" routines which may be loaded into control store along with the user microcode. By means of an interactive program in the BCPL system it is possible to read and change the contents of MATHILDA resources, and invoke the user microcode. However, it is not a normal debugger, as the microcode is not changed, and no breakpoints can be set. If the user wants such possibilities, he will have to modify the code appropriately himself, which can be performed interactively for simple modifications.

In summary, the philosophy of the system is to program the setup and invocation of firmware routines in BCPL, utilizing a few primitives of the system. When implementing an extensive firmware system, individual modules may be tested in an environment programmed in BCPL, and gradually modules may be migrated down into firmware. The system is a purely experimental system, and is only used occasionally (but sometimes heavily) for "real" applications which may benefit from special functionality or efficiency established through firmware. The facilities have evolved gradually according to needs in the (small) user community, but never following a specific plan. The system functions mostly as a testbed for ideas and experiments.

3. Other User Microprogrammable Systems

A number of commercially available processor architectures are realized through firmware interpreters located in writable control stores, e.g. recent IBM processors and DEC processors. In most cases the vendors have chosen not to support user-microprogramming, and the flexibility achieved through WCS solutions is only used for ease of maintenance and technical updating, i.e. to help the vendor and not the user. Even in the case of the B1700/1800, initial vendor support for user-microprogramming was very limited, and hard to get at. Obviously tools must have been available at Burroughs at the release of the B1700 systems, but it was initially impossible for buyers to get documentation on e.g. microassemblers and system, and users had to learn parts of the system by reading listings. The B1700 line was probably not intended for user microprogramming outside Burroughs' own development groups, and only pressure from universities and research labs made the tools available to the general user community.

The B1700/1800 is an example of what might be called a self-supporting system, i.e. tools are available on the system itself. Although examples of higher-level languages for firmware development on the B1700 have been reported (e.g. [13]), the standard tool for program construction is the assembly language MIL, which contains "syntactically sugared" representations of the (vertical) microinstructions, and has macrofacilities. In the B1700 System at Utah [14] a set of macroextensions is available (SMACK) which extends MIL providing some standard pseudo-instructions, but also debug statements for tracing, and statements for interfacing to the operating system MCP, in particular the file system, through the control store resident GISMO nucleus. The B1700 system is built around the philosophy that compilers create "codefiles", which contain not only compiled code, but also necessary information that the proper interpreter can be invoked when the execution of a piece of software is requested. If a user wants to execute a piece of firmware, it has to look to the system as an interpreter, and the invocation of the microcode has to take place as if some compiler had delivered a codefile requesting the usage of the microcode as an interpreter of some target code.

In the Utah system the LOADER simulates the effect of a compiler, and creates such a codefile. It is possible to specify to the LOADER initializations of scratchpad registers, and the static data-area of the microprogram, together with file specifications. Notice that such initializations may differ from one invocation of some microcode, to another invocation of the same (shared) code. However if all invocations require some common scratchpad initializations, all codefiles must contain the proper data.

The only firmware debugging tool which seems to be available at Utah is the tracing facility, which is a physical machine equivalent of the facilities found in non-interactive (batch-type) simulators.

One of the very early microprogrammable processors which was designed for and was marketed as a user microprogrammable system, is the Standard Computer Corporation MLP900, the basic component of the PRIM system [8] at the USC, Information Sciences Institute in Los Angeles. The MLP900 is running under the control of the PRIM system on a DEC10, and is accessible on the ARPA network.

In the basic philosophy the PRIM system corresponds to the RIKKE-MATHILDA system, whenever the latter is used in the mode where RIKKE is initiating and controlling the execution of microcode in MATHILDA, and utilizing WS as a shared storage. The PRIM system is however a multiuser system and contains hardware and software to assure the protection of other

users running under PRIM, as well as other processes running under the TENEX system on the DEC10.

The TENEX system is based on a virtual memory system, and the MLP900 has been modified to contain an equivalent memory address translator, allowing a PRIM process in the DEC10 and its subordinate emulator in the MLP900 to share the same virtual memory. Also the MLP900 has been modified to be able to run in a priviledged state at the firmware level. A small resident firmware operating system in the MLP900 (the microvisor) is responsible for switching among emulators and handles service requests from emulators, as well as page faults, to be served by the MLP driver and TENEX on the DEC10.

As seen from the user, the PRIM system acts as the interface to firmware under development, as well as to the emulator user. In its exec-state, the PRIM system allows the user to define the environment of an emulator (i.e. its target machine), e.g. to associate target machine devices with DEC10 physical devices or files, to save target environments for later use, or to restore such environments.

From the exec-state the user can enter the debug-state, which can operate as a target-independent system for debugging the emulator, as well as the target machine defined by the emulator. A particular feature is the possibility to associate with breakpoints "break-time programs" who are invoked at the breakpoint, thus allowing tracing and monitoring automatically.

In the third state, the execute-state, control is in the emulator. The user terminal can only interact through "intervention characters" (e.g. a defined break), unless the terminal has been "mounted" as a device on the target machine. The TTY can act as one target input device only, but possibly as several output devices.

Emulators are created outside the PRIM system, as well as target code may be compiled in the standard TENEX environment. In the exec-state the environment is defined, possibly including appropriate symbol tables for the debugger, in the form of tables to be used by the PRIM system. In particular tables describing completed emulators, make new target systems available as tools to be used by other users.

Both the B1700/1800 and the PRIM systems have been organized to support the development of emulators, and to facilitate the inclusion of new target machines in the environment of the common user. In contrast other development systems have been reported where the systems are organized as

to allow the inclusion of new primitives, realized in firmware, to already existing target machines. Some such systems just utilizes previously un-assigned opcodes, allowing extensions to be added to the instruction set, and made available through modification of the target assembler [7]. The utilization of such added functionality is hampered by the fact that compilers have to be modified, to make the new facilities (or extra efficiency) available in high level languages.

Another and more flexible approach has been reported in [9]. A CAL Data 135 is used as the host of an emulated PDP11/45, which supports the C-language and the UNIX operating system [18]. A few unassigned opcodes have been used to add the primitives necessary to load and call upon user developed firmware, and these primitives have been made available in C also. The system distinguishes between static (resident) and dynamic (overlaid) firmware, where user created microcode falls in the second category, and is referenced through the file system. The system works much the same way as the RIKKE system does, and seems well suited to the migration of primitives into firmware. It has been used to improve the efficiency of C and UNIX by microcoding the function-enter and -exit operations. However it is not apparent that the system can be used to (and is intended to) support complete emulators and their target machines.

The paper also contains a discussion of the objectives, and the tools which should be available in such a system. The author seems particularly concerned that it should be possible to "authorize" firmware through a validation routine, before it can be included and called upon. However it does not seem very likely that this process can be automated in the foreseable future, as the present state of affairs for "ordinary" systems programs (e.g. operating systems) may indicate. The work on verification of firmware may take us part of the way, but assuring that a microprogram is realizing the functionality it is supposed to is one thing, but to prove that it is harmless to its environment is quite different. Any proof or verification of a program is based upon some formal specification of the (virtual) machine on which it is supposed to run, and if the program is "correct" it is (of course) only accessing resources of the specified machine. But since the specified machine may not be completely equivalent to the actual, and resources in general will be shared, interference from a harmful program (or user) may still occur, unless the whole system can be proven correct [16]. Some interesting work on verification of firmware, based on host and target machine descriptions in a hardware decription language ISPS, has been reported in [3].

We may conclude this section on user microprogrammable system by concluding that (as illustrated particularly by the PRIM system) quite sophisticated systems do exist, however they have been developed mostly by users, and not by vendors. Even for the QM-1 [19] there seems to be a need for users to create their own development system [6]. Many systems which have been reported on are however at the other extreme, consisting only of assemblers/simulators, which is fine for teaching purposes, in particular when the simulator is interactive. But when the firmware is to be integrated in the system, the user is left with deadstarts, lights and switches.

4. Dedicated Control Systems

In this section we will restrict the discussion to the development of firmware for systems whose underlying hardware architecture changes from one system to the next, i.e. where the control unit is designed to fit a particular application. It might be a control unit of a bit-sliced processor, or that of a more general controller, where the firmware is to reside in ROM storage.

By nature, some sort of cross support is needed when developing firmware for ROM based control. As demonstrated in the previous sections, cross support for firmware development may be very advantageous. During development, the read-only control store can be substituted by an equivalent writeable storage, which can be loaded from the supporting system. With suitable monitoring of the control unit under debugging, and possibly by some simulation of the environment of the unit, the development system can be utilized for debugging and testing of the firmware before it is being "cast" into its final ROM storage. Such systems have been commercially available for use with standard microprocessors, to develop software which is to end up in ROM storage for a particular fixed application.

Lately such systems have also become available to be utilized in the development of firmware for bit-sliced control units, e.g. the Motorola Exorciser system modified to support systems built on the M2900 or M10800 series of components [20]. Besides a primitive facility for entering bitpatterns into the fields of the simulated control store words (in hexadecimal), the system described in [20] contains a debugger with facilities for executing the program with breakpoints. Displaying is limited to the contents of the current microinstruction, but the system under development may itself contain additional displays. The development system as described in [20] is obviously quite primitive, in particular with respect to the tools available for the

programming, but an improved system (MAZE29/800) has been described in [1], which allows for the definition and use of an assembly language, and source editing.

The obvious problem with such "on-line" tools is the limited amount of knowledge the development system has on the system under construction. The minimal knowledge required, in a system as discussed above, is information on the structure of the control store, i.e. its width and length. During debugging, any reference to the (simulated) control store has to be checked against the set of defined breakpoints, to check for the break condition. If this is performed in software obviously this may cause problems in time-critical applications, but could possibly be avoided using a set of hardware comparators (e.g. a small associative memory).

In order that the debugging system can inform the user on the state of the system, suitable probes may be connected to provide a feed-back of information. However such a solution does not seem more flexible than adding display-panels to the system under debugging. There will probably be major parts of the state vector of the system which will be inaccessible for interrogation, unless special-purpose display firmware is developed, loaded into control store, and called upon during debugging. And it does not seem very likely that such display firmware routines can be constructed automatically, even if the development system has a very detailed knowledge on the system under testing. The same problem arises with respect to interactive corrections of the state vector during debugging.

The situation is very different with the firmware construction. Sophisticated assemblers and even possibly higher level languages can be generated automatically from syntactical and semantic (architectural) descriptions, if sufficient computing power is available in the development system (e.g. [25]). But most often for cost-reasons "on-line" development systems are based on (dedicated) microprocessor systems, which only provide little computing power, in particular due to very limited memory space.

The most advanced development and test systems have so far been pure "off-line" systems, based on simulation of the system under design, and implemented on medium to large scale computer systems. In [15] a microassembler and simulator generator system has been described. It does not use a hardware description language (HDL), but an ordinary programming language Simula, which is used to build descriptions of the hardware components, which in turn are used in a description of a system. Simula is a discrete event simulation language, and as such well suited for a proper

simulation of the system under design. The simulation could possibly even include the environment, with proper timing. With a library of component descriptions, the individual architecture design may be simulated, and can be checked out before physical construction.

The use of "proper" hardware description languages in the field of microprogramming has been discussed in [5], based on experience with a particular language ISP', in automatic construction of simulators for host processors, to describe target architectures and microcomputer networks. Also the application of such languages for more general communication purposes, and in verification is discussed.

There is no doubt that it is possible to create very sophisticated "off-line" systems, which (when properly parameterized) can provide excellent tools for the development, simulation/debugging and verification of firmware for a "custom-designed" control unit.

5. Conclusions

The most sophisticated firmware development systems have no doubt been constructed for the utilization of user microprogrammable host architectures. The most flexible systems seems to be the self-supporting (e.g. B1700, QM-1 and RIKKE), which potentially are as well suited to support migration of primitives into firmware, as to support new target machines through interpretation. The cross-supported systems (e.g. PRIM and RIKKE/MATHILDA) are equally well suited for emulators, and provide interesting extra possibilities for utilizing the true parallelism. The tools which are or potentially may be available in development systems for both classes of systems, do not seem to make one sort of system preferable to the other.

The most serious problem with improving the tools for user microprogrammable systems is to handle the problems of horizontal instructions and parallelism in compilation and verification of firmware. Another problem is in the protection of innocent users and systems against malevolent firmware (or programmers), where the hosts usually provide little or no facilities for protection or virtualization.

The systems available to aid in firmware development for "custom-designed" control units are much less satisfactory. Only in pure "off-line" (simulation) systems can reasonable systems be generated, by means of formalized

descriptions, to fit a particular host design. Unfortunately such development system generators can only be implemented on quite powerful computing systems, which may be prohibitive for many development groups, considering the potential applications of bit-sliced control units.

The more economically feasible (microprocessor based) "on-line" development systems have inherent limitations when applied to such "custom-designed" control units. The programming part of the firmware development can easily be facilitated, but is seems difficult to provide full scale debugging facilities, as well as proper verification, in such systems.

References

[1] T. Balph, W. Blood: "Assembler Streamlines Microprogramming". Computer Design, Dec. 1979.

[2] S.E. Clausen: "Optimizing the Evaluation of Calculus Expressions in a Relational Database System". Aarhus University, DAIMI PB-97, April 1979, to appear in Information Systems, 1980.

[3] S. Crocker, L. Marcus, D. vMierop: Machine Description and Verification Technology, Microcode Verification Project: Interim Report. USC, Information Sciences Institute, ISI/WP-13.

[4] S. Davidson, B.D. Shriver: "An Overview of Firmware Engineering". Computer, May 1978.

[5] P.J. Drongowski, C.W. Rose: "Applications of Hardware Description Languages to Microprogramming: Methods, Practice, and Limitations". Proc. Micro 12, Sigmicro Newsletter vol. 10, no. 4, Dec. 1979.

[6] C.W. Flink: "EASY - An Operating System for the QM-1". Proc. Micro 10, Sigmicro Newsletter Vol. 8, no. 3, Sept. 1977.

[7] F.F. Fung, W.K. King: "The Implementation of a User-Extensible System on a Dynamically Microprogrammable Computer". Proc. Micro 10, Sigmicro Newsletter Vol. 8, no. 3, Sept. 1977.

[8] J. Goldberg, A. Cooperband, L. Gallenson: "The PRIM System: An alternative architecture for emulator development and use". Proc. Micro 10, Sigmicro Newsletter Vol. 8, no. 3, Sept. 1977.

[9] R.K. Guha: "Dynamic Microprogramming in a Time Sharing Environment". Proc. Micro 10, Sigmicro Newletter Vol. 8, no. 3, Sept. 1977.

[10] P. Kornerup, B.D. Shriver: "An Overview of the MATHILDA System". Sigmicro Newsletter, Jan. 1975.

[11] P. Kornerup, B.D. Shriver: "A Unified Numeric Representation Arithmetic Unit and its Language Support". IEEE-TC, Vol. C-26, no. 7, 1977.

[12] E. Kressel, E. Lynning: "The I/O Nucleus on RIKKE". Aarhus University, DAIMI MD-21, Oct. 1975.

[13] J.B. Marti, R.R. Kessler: "A Medium-Level Compiler Generating Micro Code". Proc. Micro 12, Sigmicro Newsletter Vol. 10, no. 4, Dec. 1979.

[14] E.I. Organick, J.H. Hinds: "Interpreting Machines: Architecture and Programming of the B1700/1800 Series". North-Holland Publ. Co., New York, 1978.

[15] M. Persson: "Design of Software Tools for Microprogrammable Microprocessors". Proc. EUROMICRO 79, in "Microprocessors and their Applications". North-Holland Publ. Co. (1979).

[16] G.J. Popek, D.A. Farber: "A Model for Verification of Data Security in Operating Systems". CACM vol. 21, no. 9 (1978).

[17] M. Richards: "BCPL: A Tool for Compiler Writing and Systems Programming". Proc. AFIPS 1969 SJCC vol. 34.

[18] D.M. Richie: "The UNIX Time Sharing System". CACM vol. 17, no. 7, 1974.

[19] R.F. Rosin: "An Environment for Research in Microprogramming and Emulation".
CACM 15, 8 (1972).

[20] K. Schneider: "Development tools for bit-slice microprocessors".
Euromicro Journal Vol. 4, no. 1 (1978).

[21] J. Stoy, C. Strachey: "An Experimental Operating System for a Small Computer". Part I: General Principles and Structure. Part II: Input/Output and Filing System.
Computer Journal, vol. 15, no. 2 & 3 (1972).

[22] I.H. Sørensen, E. Kressel: "RIKKE-MATHILDA Microassemblers and Simulation on the DECsystem 10".
Aarhus University, DAIMI MD-29, Dec. 1977.

[23] I.H. Sørensen: "System Modelling".
Aarhus University, DAIMI PB-87, March 1978.

[24] O. Sørensen: "The Emulated 0code Machine for the Support of BCPL".
Aarhus University, DAIMI PB-45, April 1975.

[25] E. Tamura, M. Tokoro: "Hierarchical Microprogram Generating System".
Proc. Micro 12, Sigmicro Newsletter Vol. 10, no. 4, Dec. 1979.