Embedding the MatPLC

Mário de Sousa, Adriano Carvalho, Rafael Ferreira
Faculdade de Engenharia da Univ. do Porto
Dept. Eng. Electrotecnica e de Computadores
R. Dr. Roberto Frias, 4200-465 Porto
msousa@fe.up.pt, asc@fe.up.pt, ee96153@fe.up.pt

Abstract

The MatPLC is an international project to develop a program similar to a PLC (Programmable Logic Controller) for POSIX operating systems. It is an open source application consisting of a core, generic modules, and tools for creating custom modules. Since many control and monitoring systems are distributed in nature, they require that the application be running on an embedded system capable of withstanding harsh environments.

This paper describes the porting of the MatPLC code base to a commercial embedded system running on a PowerPC CPU. Time measurements were taken to determine whether the resulting code would be suitable for control applications.

1. Introduction

The technology used in PLCs (Programmable Logical Controllers) have evolved with the times, to the point that many modern top of the range PLCs are actually full fledged computers in disguise, executing modern operating systems. More importantly, and in order to take advantage of economies of scale, many PLC vendors have started to adopt hardware similar to PCs (Personal Computers).

The MatPLC project was started with the intention of eliminating the lock the vendors have on the end-users, by taking advantage of open standards as well as open source operating systems running on the de facto standard PC platform. It has currently been successfully tested on both Linux, as well as the QNX real-time operating system, both executing on standard Intel PC hardware. Most industrial communication networks are supported (Modbus, DeviceNet, Profinet, ASI, etc.), either directly or through the use of a network interface card manufactured by Hilscher.

Software standards are also being taken into account. Currently the project includes a compiler for the IL (Instruction List) and ST (Structured Text) programming languages defined in IEC 61131-3 (International Electrotechnical Commission, 2001).

Figure 1. MatPLC Architecture, showing a selection of modules in the top part and the sections of the library in the bottom.

Access to all shared resources is made through the MatPLC library routines which offer PLC-like semantics for the modules that wish to use them, such as inputs that only change at the beginning of the logic and outputs that are only written at the end of the logic. The library is divided into several sections (Fig. 1):
• configuration memory manager (cmm) - manages the shared memory area that stores core configuration data, guaranteeing that all modules share the same configuration;
• global memory manager (gmm) - manages the shared memory area used to store the state of the plc points;
• synch section - handles the synchronization between modules;
• period section - enforces scan loop timings;
• state section - handles module execution state;
• configuration section - parses the configuration files;
• log section - allows every module to produce log messages in a consistent manner. These are timestamped and written to a logging file.

A more detailed explanation of the internal mechanisms used within the more complex sections follows.

2.1. The Global Memory Map
Since the MatPLC architecture hinges on the simultaneous execution of several modules, every access to the global memory map by a specific module must be made to be atomic with respect to the other modules. Several modes are available for enforcing this constraint (Fig. 2), none of which is optimal for all possible scenarios in which the MatPLC is expected to be used. The application builder may choose any of the access modes for each executing module.

![Figure 2. Synchronization of gmm memory maps (local, isolate and shared)](image)

For the default 'local' mode, a local copy of the memory map is created for the module (Fig. 2 - Module B). When a module accesses a plc point, it is actually accessing its local memory map. Local and global memory maps are synchronized by calling the plc_update() function, which is protected by a semaphore, providing atomic updates with respect to other modules.

The second mode, 'isolate' (Fig. 2 - Module C), completely isolates the module from the shared memory used by the PLC, and is mostly used by untrusted modules (e.g. still in a debugging stage). It uses sockets to forward the plc_update() function call from the module to the proxy, introducing significant overhead.

The third and last mode, 'shared' (Fig. 2 - Module A), assumes that simultaneous access to the shared map will never occur, and therefore gives the module direct access to the global memory map with no synchronization enforced. This may be guaranteed if all modules are running a scan loop with each loop executing in turn (possible due to the synch section, described below), or if the modules access disjoint portions of the global map.

2.2. The Synch Section
The synch section allows the application builder to specify the sequence of execution of the running modules. This is achieved by specifying a Petri net, taking into account that particular synchronization points in each module (usually beginning and end of scan, but optionally others) are associated with the firing of a transition. When such a synchronization point is reached during the module's execution, the module blocks until the transition fires (assuming it cannot fire immediately). Note that a transition will not fire unless a module is waiting on it; in this the semantics differ from those of a standard Petri net.

The synch section leverages the standard SysV semaphores to simulate the synchronization Petri net, using a single SysV semaphore set, with a semaphore for each place. Each transition is implemented by simultaneously waiting on all the appropriate semaphores, a functionality of SysV semaphores not supported by POSIX semaphores.

However, two alternative versions of this library were implemented which use POSIX semaphores and POSIX mutexes for the times when SysV semaphores are not available. The POSIX mutex version is used on Real-Time operating systems (such as QNX) due to the capability of supporting scheduling protocols which limit priority inversion. The POSIX semaphore version is used when the underlying operating system does not support mutexes across processes.

2.3. The Period Section
This section enforces maximum scan rates for each module. It uses POSIX timers to set an alarm that goes off at every multiple of the desired scan period, at which time an alarm counter is incremented. When a module is ready to start a new scan it decrements the alarm counter and continues with the scan. If there are no outstanding alarms, the scan is delayed until the next alarm goes off.

2.4. The State Section
This section of the library handles RUN/STOP modes both for the MatPLC as a whole and for each module. A module will only execute a scan if both the whole MatPLC and the module itself are in RUN mode.
In order for these modes to work correctly with the synch library, they are implemented by adding hidden places to the synch petri net. These places are connected with arcs to the begin of scan synch transitions in such a way that the transition will only fire if both the PLC and the module are in RUN mode, with the tokens being replaced after firing. A module is thus able to atomically verify the conditions required by both the state and the synch sections.

3. The SixnetIO Embedded System

SixnetIO is a company based in the United States that manufactures hardware for use by the automation industry. Its portfolio includes remote terminal units (RTU), distributed control system (DCS) controllers, remote terminal Input and Output (IO) units based on Ethernet and RS485, as well as real-time industrial Ethernet switches and industrial telephone modems. All solutions are DIN rail mountable, and ruggedised for the industrial environment.

3.1. Hardware

The SixnetIO RTUs and DCS controllers are in fact embedded computers, based on a 32 bit PowerPC CPU, with a true 32 bit data bus, and running at 50 Mhz. They sport from 16 MBytes Flash and 16 MBytes Dynamic RAM memory, up to 128 MBytes Flash, 64 MBytes Dynamic RAM, and 2 MBytes of battery backed RAM. Flash RAM is used to emulate a hard disk, and is accessed as such from user programs. Battery backed RAM is used mostly for keeping logging information, as its contents are not lost between power cyclings. Access to this RAM is also made from user programs as a second emulated hard disk.

The SixnetIO terminals come with an assortment of:
- one 10/100 BaseT auto-detecting Ethernet port,
- 16 MBytes flash disk,
- 16 MBytes dynamic RAM,
- 512 kBytes of retained RAM (battery-backed),
- one 10/100 BaseT auto-detecting Ethernet port,
- 16 MBytes Dynamic RAM,
- 2 MBytes of battery backed RAM.

All software comes pre-installed on the Flash disk, occupying approximately 10 MBytes. On the other hand, the kernel and base services, while running, take up around 6 MBytes of Dynamic RAM, leaving another 10 MBytes for user applications.

The battery backed RAM is formatted as a disk, and mounted onto a subdirectory of the root filesystem. Access to this RAM is made by simply reading and writing files located on that same directory.

3.3. Development Environment

To ease program development for the platform, SixnetIO also provides a development package, called IADK (IPm Application Development Kit). This kit contains a gcc cross compiler tool chain that runs under Linux/Intel systems, and produces Linux/PowerPC binaries. Missing from this tool chain is the 'libtool' utility, used to portably develop dynamically linked applications.

The kit also includes static and dynamic versions of several commonly used libraries, including the standard C and UNIX system call libraries. The rt (Posix Threads and Timers), maths and the curses (control of text based terminals) libraries are also made available. As expected, the dynamic version of these libraries also come pre-installed on the embedded Linux platforms, where the static versions are no longer required.

The IADK kit also includes a proprietary library used to access the local digital and analogue Input/Output ports. Using this library a user program may read and/or write to local I/O by calling two simple functions. In reality, local I/O is controlled by a kernel module (device driver), and the library functions are a simple interface to that same module.

4. Porting the MatPLC

4.1. Chosen Hardware

Out of the options made available by SixnetIO, the most basic of the RTUs was chosen for the first port of the MatPLC. This choice was conditioned upon the following reasons:

- least expensive;
- if the MatPLC could be ported to the least powerful hardware, then it would also run successfully on the remaining hardware;
- this most basic RTU also included some local Inputs and Outputs which allows the development of a MatPLC I/O module, and subsequent testing of the same.

The chosen RTU, known to SixnetIO as the MiniVersaTRAK mIPm (tm) (VT-MIPM-131-D), comes with:
- 16 MBytes flash disk,
- 16 MBytes dynamic RAM,
- 512 kBytes of retained RAM (battery-backed),
- one 10/100 BaseT auto-detecting Ethernet port,
two RS232 serial ports,
one RS485 serial port,
12 Digital Inputs
8 Digital Outputs
6 Analogue Inputs

In order to have additional I/O which shall be used in the future on a demonstration platform, an Ethernet based remote terminal unit was also acquired (ET-8DI2-8DO2-H), with 8 digital inputs and 8 digital outputs. This remote terminal acts as a Modbus Slave over TCP/IP, which means that a port of the MatPLC's Modbus Master I/O module would be required.

In order to have additional I/O which shall be used in the future on a demonstration platform, an Ethernet based remote terminal unit was also acquired (ET-8DI2-8DO2-H), with 8 digital inputs and 8 digital outputs. This remote terminal acts as a Modbus Slave over TCP/IP, which means that a port of the MatPLC's Modbus Master I/O module would be required.

4.2. Cross-Compiling the MatPLC

Porting the MatPLC to the embedded system consisted of using the cross-compiler of the IADK environment to compile the main MatPLC libraries, along with the most essential modules. The first hurdle that had to be overcome was the lack of the 'libtool' utility in the IADK development environment.

The MatPLC is developed with portability in mind, having been successfully compiled on many Linux/Intel distributions, as well as on a QNX Real-Time OS. With this in mind, the dynamic libraries are not generated directly using the operating system's tools, as these tools change significantly between OSs. The generation of dynamic libraries is therefore achieved by making use of the 'libtool' utility, which hides the details of library generation from the user.

The libtool utility was therefore ported by the authors to the IADK cross-compiling environment. With this achieved the main libraries and tools all compiled successfully. However, running the basic demo from the MatPLC distribution raised several issues. The vitrine module (a simple and configurable text based user interface module based on the curses library) did not recognise the terminal being used, and shutting down the MatPLC resulted in a segmentation fault.

The vitrine module issue was the result of the lack of a terminfo database in the embedded system. This database is used by the curses library to determine the capabilities of the terminal in use. The lack of this database on the pre-installed embedded system does not make much sense as it comes with the curses library pre-installed.

The segmentation fault issue was tracked down to a bug in the 'semctl()' system call in the glibc library, and a workaround had to be used in order not to activate the bug.

With the above issues taken care of, the basic demo ran successfully. This demo, as the name describes, is very basic and consists of the vitrine module that shows four 'lights' on a terminal, and a logic 'chaser' module that turns these four lights on and off sequentially.

4.3. The Sixnet IO module

Porting the MatPLC to the embedded system would not be complete without an IO module which allows the MatPLC access to the local digital and analogue IO. Fortunately the MatPLC comes with a common IO library, which contains all the common code found in the IO modules. This library only needs to be linked to functions that access the hardware, as well as functions that parse the hardware address format from the MatPLC configuration file, in order to generate a full IO module.

The IO address format adopted was XX.YY, where XX is one of DI, DO, AI or AO, referencing Digital Input/Output, and Analogue Input/Output. YY is a number identifying the specific IO being referenced.

Access to the local IO hardware was made using the library provided by SixnetIO. Only two functions of this library were used, namely IODBwrite() and IODBread().

The resulting IO module was tested along with the basic demo, and the sequencing lights were transferred to the local digital outputs.

4.4. The Remote IO Terminal

As the remote terminal is accessed using the Modbus/TCP protocol, MatPLC's Modbus master module was also compiled to the SixnetIO platform. This module was tested successfully in conjunction with the remote IO terminal (ET-8DI2-8DO2-H).

5. Benchmarks

5.1. Set-up

Several benchmarks were obtained of the MatPLC executing on the SixnetIO embedded system. As most PLCs run in a loop - or cycle - the MatPLC was configured to also execute an infinite cycle when measuring the benchmarks. The time taken to execute each cycle was taken as the benchmark.

Direct measurement of the cycle times is not possible using either the local digital outputs, or the operating system's timers. The local digital outputs have response times in the order of 10 ms when being controlled through the library functions made available by
SixnetIO. Likewise, the resolution of the internal POSIX timers under the used 2.4.18 Linux kernel is 10 ms. This could be partially overcome by measuring the time taken to execute several cycles at a time, and then determining the average. Obviously this average may hide rather large delays that may not be compatible with the control application.

The PowerPC CPU used in the embedded system does however have an internal 64 bit register that is continuously incremented at a fixed 3,215 Mhz (1/16 th of the clock frequency). This register may be read using the assembly instructions 'mtbl' and 'mtbu', and from its values it is possible to determine the passage of time with a resolution of 0,32 µs. The scan cycle times were therefore measured with a high degree of accuracy by simply reading the current value in the register at the beginning of each cycle. The values were initially stored in memory, and then transferred to a file at the end of a 1000 scan cycle run in order not to introduce any delays in the scan cycle itself.

A variant of the basic demo was used for the purpose of toggling the local outputs to use as an example. The MatPLC was configured to execute only two modules, namely the Sixnet I/O module, and the chaser logic module, the latter module being the control logic module used by the basic demo to control a sequence of four lights. As mentioned previously, these modules were synchronised to execute one after the other, in order to more closely resemble the scan cycle of traditional PLCs.

5.2. Results

The results of the measurements are presented in the following table of figure 4. The first line are the results obtained when the MatPLC executes the scan cycle continuously, without any timer synchronisation. The following lines are the results obtained when the MatPLC is configured to execute the scan cycle at a fixed execution period (10 ms, 20 ms, and 50 ms respectively).

Each configuration was repeated 20 times, resulting in the minimum, maximum and average times presented in the table.

<table>
<thead>
<tr>
<th>Period</th>
<th>Min.</th>
<th>Avg.</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>3,97</td>
<td>5,94</td>
<td>73,9</td>
</tr>
<tr>
<td>10</td>
<td>12,1</td>
<td>20,1</td>
<td>37,2</td>
</tr>
<tr>
<td>20</td>
<td>13,6</td>
<td>30,0</td>
<td>47,0</td>
</tr>
<tr>
<td>50</td>
<td>53,4</td>
<td>60,2</td>
<td>74,3</td>
</tr>
</tbody>
</table>

Figure 4. Measured scan cycle times with MatPLC in 'local' mode (all times in ms).

Further measurements were obtained with the same configuration as previously used, differing only by the fact that the MatPLC was configured to access the common global point memory in shared mode. With this new configuration the MatPLC no longer does the synchronisation between the local and global versions of the global point map, once for each module on every scan cycle. This saves considerable time, as the global point map used for the first measurements was 8 kBytes, even though no more than a single byte was being used by the application. Note that with the configuration used (two modules running in lock-step) no advantage is obtained from using the first 'local' method of accessing the global point map. In fact, the main advantage of using the 'local' mode would be to allow the two modules to execute asynchronously.

<table>
<thead>
<tr>
<th>Period</th>
<th>Min.</th>
<th>Avg.</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0,40</td>
<td>0,65</td>
<td>22,13</td>
</tr>
<tr>
<td>10</td>
<td>13,0</td>
<td>20,0</td>
<td>38,4</td>
</tr>
<tr>
<td>20</td>
<td>21,8</td>
<td>30,1</td>
<td>51,1</td>
</tr>
<tr>
<td>50</td>
<td>56,8</td>
<td>60,2</td>
<td>73,4</td>
</tr>
</tbody>
</table>

Figure 5. Measured scan cycle times with MatPLC in 'shared' mode (all times in ms).

With the 'shared' configuration and no enforced period the scan times are very short, and can now be favourably compared to the scan times of commercial PLCs. In fact, the scan times are much shorter than the reaction times of the local outputs both for the 'shared' and the 'local' access methods. Nevertheless, even though the average scan cycle times may be short, the maximum execution times are very large compared to these (one order of magnitude for the local access method, and almost two orders of magnitude for the shared method). These results are of course related to the fact that the operating system being run is not real-time, and is not deterministic.

It is possible, however, to execute the MatPLC at a higher priority using a RT scheduling protocol (the POSIX fixed priority SCHED_FIFO method). With this method the operating system scheduler will give the MatPLC higher execution priority. However, since this is not a real-time OS, the operating system itself will still have relatively long periods of time in which it will block all processes from execution. This can be seen in the following results presented in the tables of figures 6 and 7.

<table>
<thead>
<tr>
<th>Period</th>
<th>Min.</th>
<th>Avg.</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>4,09</td>
<td>4,35</td>
<td>5,08</td>
</tr>
<tr>
<td>10</td>
<td>15,0</td>
<td>20,1</td>
<td>49,6</td>
</tr>
<tr>
<td>20</td>
<td>26,9</td>
<td>30,0</td>
<td>47,1</td>
</tr>
<tr>
<td>50</td>
<td>56,8</td>
<td>60,2</td>
<td>75,4</td>
</tr>
</tbody>
</table>

Figure 6. Scan cycle times (ms) in 'local' mode, at high priority using SCHED_FIFO.
<table>
<thead>
<tr>
<th>Period</th>
<th>Min.</th>
<th>Avg.</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0.48</td>
<td>0.58</td>
<td>1.51</td>
</tr>
<tr>
<td>10</td>
<td>16.0</td>
<td>20.0</td>
<td>50.6</td>
</tr>
<tr>
<td>20</td>
<td>22.2</td>
<td>30.1</td>
<td>50.6</td>
</tr>
<tr>
<td>50</td>
<td>52.3</td>
<td>60.2</td>
<td>73.6</td>
</tr>
</tbody>
</table>

Figure 7. Scan cycle times (ms) in 'shared' mode, at high priority using SCHED_FIFO.

From all the results it can also be seen that, when executing at a fixed period, the real execution period is approximately 10 ms longer than the requested value. This may have to do with the fact that 10 ms is also the internal clock resolution of the Linux OS. However, this behaviour is not reflected when the MatPLC executes on Linux running on Intel compatible CPUs. The suspicion therefore falls on the PowerPC port of the Linux OS, although further and more thorough analysis is required in order to come to a definite conclusion.

6. Conclusions

Porting of the MatPLC to the PowerPC architecture was rather straightforward, especially taking into account that the development environment with a cross compiler and all required libraries are made available by the vendor. The lack of the libtool utility however resulted in some time being devoted to its configuration for the new environment.

The SixnetIO hardware itself is robust, but the local I/O may be too slow for many control applications (up to 10 ms response times). However, this delay may be caused by the way the library supplied by the vendor is accessing the hardware itself, and may not have anything to do with the hardware. There is still hope that the vendor may change the way this hardware is accessed, and therefore fix this issue with a simple software fix.

Execution times of the MatPLC are sufficiently fast, even when executing on the embedded processor running at 50 Mhz. However, special attention must be paid to the extra delay being introduced when executing at a fixed period. The possibility of adding real-time patches to the kernel of this embedded system should be investigated.

6.1. Future work

This work will continue by using the MatPLC on the SixnetIO hardware to control a small demonstration manufacturing cell located in the premises of the University of Porto Engineering Faculty. This cell is composed by a CNC (Computer Numerical Control) controlled lathe, a CNC milling machine, a small automatic warehouse, and several simple robots for moving work pieces on and off the CNC machines.

A port of a Real-Time variant of Linux should be attempted to the SixnetIO hardware with a view of determining if better response times and higher predictability can be obtained.

Additionally, development of the MatPLC will continue with the aim of supporting the remaining languages of the IEC 61131-3 standard. I/O modules will also be developed as the need arises to support new hardware or communication protocols.

7. Acknowledgements

The authors would like to thank Curt Wuollet for taking the initiative of starting the MatPLC project. We also appreciate the contributions of Jiri Baum and Juan Carlos Orozco to the project.

References