Abstract—The authors have developed a mechatronic tracking system controlled by a hard real-time embedded Linux based system. The control software is implemented on a standard PC/104-Plus embedded single board computer (CoreModule 745 with Intel Atom processor). The paper is focused on various aspects of developing an embedded version of Linux software that are mostly not dependent on the application. The reader is expected to be an experienced Linux user.

Keywords—Embedded systems; hard real-time; Linux; software development.

I. INTRODUCTION

We have developed an embedded system for multi-camera real-time recognition and tracking of moving objects. Many algorithms of the system were described in our previous papers [1]–[8]. We focus on internal structure of the system’s software, developing tools and processes of locating the software on a destination hardware in this paper. The aim is to provide a tested way of developing an embedded control system for mechatronic systems. We want to keep our text to be "application and equipment independent" so mostly general recommendation is given.

The destination hardware is an industrial PC (PC/104-Plus or PCI/104-Express). This family of single board computers usually contains CPU, RAM memory slots, graphics controller, disc controller, Ethernet adapter, USB and serial interfaces, and some GPIO pins. To control our mechatronic system, we use the CoreModule 745 board by ADLINK company. This board employs Intel’s Atom CPU and also has a solid state disk (SSD) we call the internal disk. Its capacity of 4 GB is sufficient to accommodate the embedded software but is inadequate for a software development system.

Our mechatronic system has to communicate with a day light GigE Vision camera connected via an Ethernet adapter. A night (infrared) camera uses a frame grabber to deliver frames. We also employ IP cameras for another project. Servos and few other peripherals are controlled by CAN bus. We have added three additional board to the industrial PC computer that are connected via the PC/104-Plus bus:

1) CAN bus controller for communication with servos and some other peripherals like a control panel.
2) Frame grabber to get digital frames from the analog infrared camera. There are some GPIO devices on the frame grabber that is used to control synchronization of the day light camera, setting dimming and contrast of the infrared camera, setting monitor brightness and contrast etc.
3) A custom design board for synchronizing exposure timing and exposure duration of the daylight camera. It is required to support image processing algorithms we have implemented. [1], [2], [8].

II. BUILDING DEVELOPMENT SYSTEM

A. Linux Distributions

There are many Linux distribution. Some of them are more popular than others. According to [9], Linux Mint [10], Ubuntu [11] and Debian [12] are the most popular. There are also lightweight Linux distributions that use relatively few computer resources, e.g. Damn Small Linux [13], Basic Linux [14] for outdated computers, Alpine Linux [15].

It is possible to install a custom built Linux if necessary [16]. This book is very helpful, if deep knowledge of some Linux utilities is needed. Relations among various utilities are covered, too.

If the destination hardware is a modern industrial PC (PC/104-Plus or PCI/104-Express with a 64-bit processor), there is no reason to use a lightweight Linux distributions. There is probably no reason not to use your favorite Linux distribution. It is always possible to install required software tools and packages additionally. Having tested several Linux distributions, we have opted for the Fedora Linux distribution [17]. It is just a question of personal preference.

B. Hard Real-Time for Linux

Linux, like other general purpose operating systems, supports running soft real-time applications. The soft real-time means that software deadlines are almost always met but if not, it is not fatal. Hard real-time has stronger requirements. Anyway, the border between hard and soft real-time is a bit fuzzy.

There are several software projects providing hard real-time capabilities to Linux kernels.

The authors of the CONFIG_PREEMPT_RT patch set try to minimize the amount of kernel code that is not preemptible while also minimizing the amount of code modified by their patch. Introduction and overview can be found in [18]. [19] contains all essential links to project resources. The project is kept up to date and patches for most recent Linux kernels are available.
Xenomai [20] is, according to its authors, “a real-time development framework cooperating with the Linux kernel, in order to provide a pervasive, interface-agnostic, hard real-time support to user-space applications, seamlessly integrated into the GNU/Linux environment.” In 2003 it merged with the Real-Time Application Interface (RTAI) [21], but it became independent in 2005. We have not tested the Xenomai framework.

RTLinux (Real-Time Linux) [22], [23] is a hard real-time operating system microkernel that runs the entire Linux as a separate preemptible process. The microkernel runs several hard real-time processes as scheduled, and if some CPU time is left, it is given to Linux. We used RTLinux for robot control [24]. We stopped using it after it got commercial.

RTAI (Real-Time Application Interface, [21], [25]) lets the users to write applications with strict timing constraints for Linux operating system. Like Linux, RTAI is software written by community. It is based on the Linux kernel and makes it preemptible. Features of hard real-time system are added. RTAI traps peripheral interrupts and if needed sends them to Kernel kernel. Like RTLinux mentioned above, it runs Linux as a process if CPU time is not required by a hard real-time process. RTAI uses Adeos (Adaptive Domain Environment for Operating Systems, [26]) that provides a certain kind of hardware abstraction level. Adeos operates between computer hardware and the operating system that runs on it. Adeos is inserted into the Linux kernel as a loadable kernel module. Using of Adeos by RTAI seems to help to settle down patent disputes between RTLinux and RTAI people.

We use RTAI to develop our hard real-time software.

C. Versions of the Developed Software

We have developed software aimed towards real-time recognition and tracking of moving objects. From the user’s point of view, it is an embedded software system located on the internal disk of the destination computer. There are some additional peripherals connected to the destination computer that were listed in Section I. There is also so called development disk (usually an USB disk) employed for development and production purposes. The word Software (capital S) will be used for the software that was written by us.

The development disk contains all tools necessary to develop the Software. This means the Linux operating system, distribution Fedora [17], with hard real-time support (RTAI) added, drivers for non-standard peripherals provided by their producers, and the Software. Fedora was configured as a software development tool during installation.

The development disk is used for developing the Software on the destination hardware. If the disk is connected to the destination computer, the operating system installed on the disk boots up after powering up. The operating system together with the Software tools creates the embedded version of the Software and transfers it to the internal disk of the destination computer. After shutting down, the development disk is disconnected. Having re-started, the computer runs the embedded version of the Software from the internal disk.

We found useful to build three versions of executable from source files of the Software:

1) The debugging version. It runs only on the development disk under X Window System environment and does not use hard real-time environment. Symbolic debuggers like kdbg [27] can be used without restrictions. Graphical user interface is made as an X window.

2) The real-time version. It runs only in hard real-time environment without X Window System. It has got the final graphical user interface. Normal debugging is impossible but it can run the diagnostic subsystem described in Section III-E. The diagnostics supports time measurement functions and some rudimentary debugging of problems that cannot be solved within the debugging version. While the real-time version does almost the same as the final embedded version, it allows to go much faster through cycle of modifying the source texts, making the executable, running and testing the executable and checking the results.

3) The embedded version [7]. Actually it goes far beyond just making the executable for the embedded system. It includes making file systems on the destination computer’s internal disk and copying the necessary files into them. The embedded executable is just one of these (roughly tens) files. The diagnostic subsystem can be linked to the embedded version. It provides the only tool to find a problem, if there is something wrong with the embedded system. The diagnostic subsystem is expected to be excluded in the final embedded system.

D. Installing Development System on Destination HW

We were faced with the task to install the developing system on two different kinds of machines: a standard PC and the destination hardware. When our project was kicked off, the destination hardware was not available yet. Later, development of the peripherals for the destination hardware, listed in Section I, was in progress. We had to install the development system first on a standard PC and to start developing the Software on it. Several special peripherals are not available for a standard PC. This problem was overcome by writing software simulating the missing peripherals. This topic is covered in Section III-D in detail.

If RTAI is chosen as the hard real-time support tool, it will be probably impossible to install it together with the newest release of your favorite Linux distribution. We had to download the RTAI software first and to check which version of the Linux kernel it supports. A Linux distribution, that can work with a Linux kernel supported by RTAI, has to be installed. This usually means to be one or two releases behind the newest one for our favorite Fedora Linux distribution.

It is possible to run both debugging version and real-time version of the Software on a standard PC if the hard real-time support is installed and simulation of missing peripherals is implemented in the Software. Even the embedded version of the Software can be generated and stored on an USB flash disk or a memory card. It can be even run, if the PC can boot from this device. Modern PCs usually can. If the PC and the destination computer use substantially different CPUs, installing and running the embedded version on the PC is probably not worth the effort.
Installing any modern Linux distribution on a standard PC is a relatively easy task. Installing a hard real-time support is mostly more difficult because it can include modifying (patching) kernel source texts, configuring, compiling and installing a new Linux kernel.

To install a Linux distribution and hard real-time support on an industrial PC (PC/104-Plus or PCI/104-Express), a hard disk (the development disk) is needed and a device with installation medium has to be connected temporarily. Modern industrial computers have support for SATA and USB hard disks. We opted for an external USB connected DVD unit for installation purposes and an external USB disk as the development disk. An external USB disk is very portable and if small enough, no external power source is required.

We have three different Linux kernels on the development disk:

1) A “normal” kernel is a product of Linux installation. It is used for general purposes. The debugging version of the Software can be run under it.
2) A kernel supporting the hard real-time and including drivers of all special peripherals. Both debugging and real-time versions of the Software should be capable to run under this kernel.
3) A kernel for the embedded version. It can run the embedded system only. On the other hand, the embedded version is not expected to run on other kernels. Obviously, the kernel includes the hard real-time support. We gave some special recommendations for this kind of kernel in [7].

At least, the first two kernels have to be installed on the development PC, too. Of course, the PC may not have some special peripherals installed. If so, peripheral simulators will do the job.

We do not give any recommendation about software development kits. It is a question of personal preference.

III. INTERNAL STRUCTURE OF THE SOFTWARE

A very rough scheme showing how the software works can be shown as follows:

```c
{  
  InitializeEverything();  
  while (!quit)  
    {  
      WaitForClockSignal();  
      ExecuteOnePeriodJob();  
    }  
  ShutDownEverything();  
}
```

It is much more complicated in reality. During start-up, the Software checks which peripherals are present and initializes them. The peripherals, that are not present, are simulated by software tools. A text file containing user preferences and values for some adjustable parameters is read and interpreted. The software starts threads described in Section III-B. It switches to the hard real-time mode and the hard real-time timer is started.

A. Real-Time Scheduling

The real-time scheduling of the Software consists of three levels:

1) The foreground thread. It is a hard real-time thread running periodically with period 1 ms. We call this period the small period.
2) The background thread. It is a hard real-time thread running periodically with period 100 ms. We call this period the big period.
3) Miscellaneous or just non-real-time threads. There are several of them. They are listed and described in Section III-B. They are not subjected to hard real-time scheduling. Their priority is always lower than priority of any real-time thread.

To describe it shortly, the foreground thread makes the background thread ready to run. Having finished one period jobs, the background thread stops itself and waits for permission to continue sent from the foreground thread.

Most of the CPU time is consumed by image processing algorithms [1], [2] executed by the background thread. To avoid risk of missing the deadlines, we process only a part of picture frames provided by cameras. It is a rectangle whose center is the center of the frame. Depending on the character of the frame and power of available CPUs, the processed part of the frame can be increased or decreased adaptively from one period to the next one. As a rule of thumb, we prefer that the background thread consumes around 80% of the available time. Some CPU time is always left to non-real time threads.

We have implemented a step mode into our Software to make debugging easier. One step corresponds to one big period. Implementation of the step mode is difficult because there are several real peripherals that cannot be run in step mode. For example, IP cameras can take seconds to start or stop. Having started, they send frames that must be received as soon as possible. Other peripherals require regular communication; once being started it is necessary to send them messages and receive messages from them. The communication with such peripherals must run even if the Software is stopped in the step mode. For these reasons, the step mode cannot provide perfect stepping in any situation. In spite of this, we found the step mode useful and its implementation was worth the effort.

The debugging version has no real-time scheduling. It is organized in a slightly different way than both real-time versions. In spite of this it is very helpful when finding bugs in general algorithms is needed.

B. Threading

The basic structure of the threads has been already shortly mentioned in Section III-A.

The foreground thread is a hard real time thread running periodically with period 1 ms (the small period). It is scheduled as the highest priority thread. The most relevant tasks of one small period are:

- It activates (makes run-able) the background thread at the right time. Scheduling of the background thread depends also on the step mode.
- It monitors if the background thread keeps its deadlines. The foreground thread has the highest priority and it always runs. Due to high frequency (1000 Hz), its code is simple and less prone to bugs. If the background thread or any non-real-time thread takes more time than expected, the foreground thread can interrupt it.

- It communicates with peripherals if very exact timing within the big period is required. For example, a synchronized day light camera requires exact exposure timing and duration. This is achieved by controlling a set of GPIO signals.

- During start-up stage it is the only running thread. It starts all other threads and switches to the hard real time mode.

- During shutdown stage it switches the hard-real time mode off and waits until all other threads are finished. Then it executes the final jobs.

The background thread is a hard real-time thread running periodically with period 100 ms (the big period). It is scheduled as the second highest priority thread. The background thread is responsible for agenda to be done within one big period. The most relevant tasks to be done during one big period are:

- Communication with peripherals that need to be serviced synchronously at the start of a big period, e.g., servos. This includes fetching frames from cameras to process. Tens of milliseconds elapse between end of cameras’ exposure and finishing transfer of the frames to the computer memory. The job is done by camera drivers. This means that frames captured at the start of the previous period are processed at the current period.

- Inputs from various peripherals are processed. Artificial objects could be added to frames fetched from the cameras here (see Section III-D). The frames are processed and objects of interest are located in the frames. The locations are transformed in respect to coordinate systems independent on camera locations. The new locations are compared to locations of objects from previous big periods. We try to reconstruct movements of the objects nearby the center of camera’s field of view [2].

- Based on the results of previous operations and user actions, we construct a frame that will be displayed on the monitor during the next big period. Such frame consists of frame acquired by a chosen camera in previous period and a set of icons. More details about the graphical user interface can be found in Section III-C.

- Preparing outputs for peripheral devices.

- Simulators of peripherals, that are not present, are run.

- The prepared outputs, including the frame to be displayed, are sent to corresponding peripherals.

Miscellaneous threads are run in the spare time when no real-time thread is scheduled to run. As explained in Section III-A, we prefer to keep around 80% of CPU time for hard real-time threads and the rest is either as a reserve for them or for miscellaneous threads to run. Generally, a non-real-time thread can last many big periods and it is mostly not synchronized with the big period. These threads we be divided into two groups:

1) Threads servicing IP cameras
2) Threads servicing rarely occurring events

**Threads servicing IP cameras.** The number of the threads corresponds to the number of the IP cameras. Implementation of communication with IP cameras on a low level requires some knowledge of Real Time Streaming Protocol (RTSP), [28] to establish and close down a data delivery session with an IP camera. The Real-time Transport Protocol (RTP) [29] is employed for transmitting real-time video data. The RTP Control Protocol (RTCP) [29] is a sister protocol of RTP that provides out-of-band statistics and control information for an RTP flow regularly. Timestamps for periodic clock synchronization for the IP camera clock and the Software clock are the most important part of the RTCP protocol. To save data bandwidth, IP cameras usually use a video compression format. Several video compression formats are supported by our IP cameras but H.264 is the most important one. RTP payload for H.264 video is specified in [30]. We use the FFmpeg software package [31] to decode the H.264 video format. Several decoded video frames with their time stamps are available for the background thread to choose from. Outdated frames are dismissed.

**Threads servicing rarely occurring events.** The events are initiated by a user. There are only two. The first one is formatting and writing a new user preference file. It can last up to several seconds. The second one is computing and saving a set of numerical data that makes another user preference file. Computing a new set of data can last several minutes. If any of these two threads is running, many algorithms normally executed during a big period are restricted to get more CPU time for the miscellaneous threads.

Intel Atom processor of the destination hardware is a dual-core processor so the question of making some of our algorithms parallel arises. We see only little opportunities to modify our algorithms in such a way. We had also strict deadlines in our project. To use capabilities of the symmetric multiprocessor system, we work very carefully with priorities of individual threads and give permission to some threads to run on more than one core.

### C. Graphical User interface

The first thing to consider is whether the final embedded system is going to be equipped with a monitor. Even if the answer is negative it may be worth to maintain a GUI for development and service purposes. For a real-time embedded system with a monitor, X Window System is no good choice. It is too large and startup time is essential for some of our applications. Problems with real-time scheduling caused by X Window system were reported, too. On the other hand, we may need to run the debugging version of our application under X Window System because many nice to use debuggers are just GUIs of gdb [32]. We prefer a GUI development tool that can be used for all versions of the Software listed in Section II-C.

Some papers about choosing a GUI library for an embedded system can be found [33], [34], [35] is quite old now but
still useful. We have tried DirectFB [36] software but failed to use it with static libraries. We discuss problem of static and shared libraries for an embedded system in [7].

Now, we employ General Graphics Interface (GGI) [37]. The last version of GGI libraries was released in January 2007. It supports both an X Window System GUI for the debugging version of the Software and a small footprint GUI for the real-time and embedded version of the Software. It also provides interface for a keyboard and a mouse. The USB keyboard is not a part of the final embedded system but it can be attached and detached at any time to/from the system for debugging and service.

We found most drawing functions of GGI too slow for our purposes. To avoid this problem, we developed a system of drawing icons directly into the video memory. GGI is used only for creating a an X window (the debugging version) or setting the video card into the required mode (the real-time and the embedded version). Then the address of the video memory for both the X window and the small footprint GUI is acquired.

We keep two frames of GUI. The first one is in the video memory and can be seen on the monitor. The second one is just a chunk of RAM of the equal size. During a big period, a new frame to be displayed on is build in the RAM from the scratch. It consists of a new frame from a camera that is partially overlaid by a set of icons. At the end of the big period, the new frame is copied from the RAM to the video memory.

D. Peripheral Interface

When both a new destination hardware and the software for it is being developed at the same time, it is useful to write simulators of peripherals that are not available yet. There is one more reason to do it. Good simulators of peripherals allow to develop software on standard personal computers that are not going to have non-standard peripherals of the destination hardware. Of course, there is no need to simulate commonly available peripherals.

During start-up stage, the Software checks which peripherals are present. The peripheral simulators are engaged only for peripherals that are not present. Use of simulators can be enforced by options in a user preference file.

Simulation of peripheral is also very useful for debugging general algorithms. Many real peripherals cannot be really stopped, when step mode is on. Dry run communication with real peripherals can be hard to arrange if a general debugger has to be used. A well written software simulator has no such restrictions.

There is one good rule to follow when writing peripheral interface for a real-time system. Time stamps have to be added to every input from peripherals. Many hard to trace bugs are caused by using peripheral inputs with wrong time stamps. Some inputs (e.g. servo positions) may need buffering, too.

We found simulation of the following peripherals essential:

1) Servos. It includes simulation of the communication with the servos and simulation of their behavior. The simulator has to provide a sequence of the “actual” positions as a reply to a sequence of wanted positions.

2) Laser range finder (LRF) used to measure distances for some applications. The simulator is sent a sequence of LRF commands and it returns replies to them including time delays corresponding to the real LRF.

3) Pan and tilt cameras. We describe simulation of cameras bellow in more detail bellow. The camera simulator is called the image generator.

Image processing is a substantial part of our Software. Our algorithms are described in [1], [2], [8]. Now, we will pay attention to developing tools.

Even high end digital cameras run under ideal conditions (static cameras working in stable light conditions) give different frames after each exposure. Image processing algorithms should be resistive to it but stable images are more suitable for developing and testing algorithms. At the early stages of our project, the destination cameras were not available. We had to develop a testing environment for image processing algorithms.

The tool is called the image generator for pan and tilt cameras. It provides a different images for different pan and tilt positions. It is based on a landscape photograph that has a strip of the sky on its top. The size of the photograph (in pixels) is larger than the camera’s image size. We make six different “tiles” from the original photograph. Let us denote them: $L_{00}$ is the original photograph. $L_{0v}$ is the vertical flip of the original photograph. $S_{00}$ is the sky strip from the original photograph (the original sky strip). $S_{hv}$ is the vertical flip of the original sky strip. $S_{hv0}$ is the horizontal and vertical flip of the original sky strip. $S_{hv}$ the vertical flip of the original sky strip.

The tiles are arranged in the following way:

\[
\begin{matrix}
S_{00} & S_{0w} & S_{00} & S_{0v} & S_{00} & S_{0w} \\
S_{h0} & S_{hv} & S_{h0} & S_{hv} & S_{h0} & S_{hv} \\
S_{00} & S_{0w} & S_{00} & S_{0v} & S_{00} & S_{0w} \\
S_{h0} & S_{hv} & S_{h0} & S_{hv} & S_{h0} & S_{hv} \\
L_{00} & L_{0v} & L_{00} & L_{0v} & L_{00} & L_{0v}
\end{matrix}
\]

The number of columns in the above matrix like composition must be even but there are no other restrictions. The width of the composition determines the number of pixels used to cover the full (360°) pan circle of the simulated camera. The number of rows is given by required range of the tilt.

Being given camera pan and tilt coordinates, it is easy to compose a corresponding frame. There is no need to keep the whole matrix like composition in the memory. During start-up stage, just some of the tiles are stored in the memory. Unlimited pan rotation is allowed.

The image generator works with different typical landscape photographs, e.g. summer or winter country, city, mountainous country, sky with different types of clouds etc.

To test image processing and tracking algorithms we developed a subsystem of so called artificial objects. Having received a frame from a real camera or from the image generator described above, we can add artificial objects to the frame before processing it. The objects can move. Their type, initial positions, speed and acceleration coordinates are taken
from a user preference file. The position of the artificial objects can be reset (i.e. returned to their initial positions) at any time.

E. Diagnostic Subsystem

It is virtually impossible to use any kind of debugger when the Software runs under hard real-time constrains. There are some bugs that appear only in the real-time and embedded versions of the Software. Duration of some important software routines must be measured on various kind of hardware, too.

To overcome these problems, we introduced so called diagnostic C language macros. Depending on the definition status of an identifier, the definition of a diagnostic macro is either empty (no code is produced) or non-empty. C language directives #define and #undef are used to decide which part of diagnostics will be really engaged.

If non-empty, a diagnostic macro employs snprintf() function to format the diagnostic information. The resulting string is copied to a large buffer. If the buffer is overfilled, the oldest information is replaced by the newest one. If the Software executable is being finished correctly, content of the buffer is written to a text file in the chronological order. Obviously the diagnostic system must be designed as thread safe.

IV. EMBEDDED VERSION OF THE SOFTWARE

This topic has been already covered in full detail [7]. The paper is available on-line.

V. CONCLUSION

We have presented our experience obtained during development of the embedded software based on a real-time version of the Linux operating system. Various versions of the Software have been used to control several mechatronic systems. The realization output is oriented both toward the civil and the military sector in cooperation with a Slovak producer of the camera head orientation servos. The presented software tools and concepts are believed to be useful for authors of other mechatronic system control software based on Linux.

ACKNOWLEDGMENT

The research presented in this paper is partially supported by national grant agencies SRDA under the grant No. 0261-10 and VEGA under contract No. 2/0194/13.

REFERENCES


