



Information Technology for European Advancement

Task 1.1 – Common Characteristics with Attributes and Metrics (D.1.1.5)

Version 01 - Public
Edited by Luc Vandormael

Software Development Process for Real-Time Embedded Software Systems (DESS)

ITEA COMPETENCES involved:

- 1) Complex Systems Engineering**
- 2) Communications**
- 3) Distributed Information and services**

February, 2001

Table of Contents

| | |
|--|----------|
| PURPOSE OF THIS DOCUMENT | 2 |
| DESCRIPTION OF THE COMMON CHARACTERISTICS AND ATTRIBUTES..... | 3 |
| 1.1 REAL-TIME EMBEDDED CHARACTERISTICS | 3 |
| 1.1.1 <i>Embedded system</i> | 3 |
| 1.1.2 <i>Real-Time system</i> | 3 |
| 1.1.3 <i>Memory Constraints</i> | 4 |
| 1.1.4 <i>CPU constraints</i> | 5 |
| 1.1.5 <i>Bandwidth Constraints</i> | 5 |
| 1.1.6 <i>Power Consumption Constraints</i> | 6 |
| 1.2 FUNCTIONAL CHARACTERISTICS | 6 |
| 1.2.1 <i>Communication</i> | 6 |
| 1.2.2 <i>User Interface</i> | 7 |
| 1.2.3 <i>Command and Control</i> | 7 |
| 1.3 OPERATIONAL CHARACTERISTICS | 7 |
| 1.3.1 <i>Quality of Service</i> | 8 |
| 1.3.2 <i>Dependability</i> | 8 |
| 1.3.3 <i>Availability</i> | 8 |
| 1.3.4 <i>Reliability</i> | 9 |
| 1.3.5 <i>Safety</i> | 10 |
| 1.3.6 <i>Robustness</i> | 10 |
| 1.3.7 <i>Testability</i> | 10 |
| 1.3.8 <i>Maintainability/Serviceability</i> | 11 |
| 1.3.9 <i>Security</i> | 11 |
| 1.3.10 <i>Field Loadable SW</i> | 12 |
| 1.3.11 <i>Configurable SW</i> | 12 |
| 1.3.12 <i>Flexible SW</i> | 12 |
| 1.4 SW ARCHITECTURE CHARACTERISTICS | 12 |
| 1.4.1 <i>Software Architecture</i> | 12 |
| 1.4.2 <i>Scheduling Type</i> | 13 |
| 1.4.3 <i>Interrupt handling</i> | 13 |
| 1.4.4 <i>Tasking and Exception Handling</i> | 13 |
| 1.4.5 <i>Processor privilege usage</i> | 13 |
| 1.4.6 <i>Languages used</i> | 13 |
| 1.4.7 <i>Modularity</i> | 14 |
| 1.4.8 <i>Portability</i> | 14 |
| 1.4.9 <i>Configurability</i> | 14 |
| 1.4.10 <i>Operating Systems and Kernels used</i> | 14 |
| 1.5 HW ARCHITECTURE CHARACTERISTICS | 14 |
| 1.5.1 <i>Processors</i> | 14 |
| 1.5.2 <i>Memory</i> | 14 |
| 1.5.3 <i>Internal Buses</i> | 15 |
| 1.5.4 <i>Communication Buses</i> | 15 |
| 1.5.5 <i>Other Outside Connections</i> | 15 |
| 1.6 DEVELOPMENT TOOLS..... | 15 |

List of Figures

| | |
|---|---|
| Figure 1: definition of time-constraint metrics | 4 |
|---|---|

Purpose of this document

1. The purpose of this document is to establish the common characteristics of real-time embedded software development. This set of common characteristics will be based on the individual contributions of the WP1.1 partners. Where appropriate, some definitions and concepts, which are in widespread use in specialized literature, were included as well. Feedback from workshops, conferences and private discussions will also be taken into account.

Description of the Common Characteristics and Attributes

1.1 Real-Time Embedded characteristics

1.1.1 Embedded system

1.1.1.1 Common characteristics

The systems, for which this project aims to develop a software methodology, are indeed embedded systems. The types of systems the DESS WP1.1 partners are developing are all dedicated to specific tasks. In some applications, the complete system even consists of multiple embedded systems. The user can only access what the application allows him to. Access to the underlying operating system (if any) is never granted.

1.1.1.2 Generic definition

An embedded system is a system where the user can see the functionality but has no access to the underlying software technology.

1.1.1.3 Metrics

If a system fits the generic definition, it will be considered embedded.

1.1.2 Real-Time system

1.1.2.1 Common characteristics

The real-time aspect of the embedded systems becomes apparent in many ways. Depending on the type of application, emphasis is put on delivering a service, the quality of the service, or a certain speed in responsiveness in delivering the service.

1.1.2.2 Generic definition

Usually, three types of “real-time” are distinguished:

- Hard real-time;
- Soft real-time;
- Statistical real-time.

Hard real-time means that the deadlines **must** be met, and the overall system behaviour must be deterministic. If this is not the case, then the embedded application will be considered “failed” and it will most probably be excluded from use. This is usually the case for safety-critical systems.

The soft real-time constraint allows that deadlines are missed, however in this case the quality of service comes into play. If the quality might be endangered in such a way that the overall result becomes unacceptable, the system may be considered “failed” as well.

In statistical real-time, deadlines may be missed, as long as they are compensated by faster performance elsewhere to ensure that the average performance meets a hard real-time constraint. To be able to fully assess the consequences of the statistical behaviour, stochastic analysis is required. However, it is always possible to transform this into a deterministic analysis by investigating the worst case situation

1.1.2.3 Metrics

Test equipment will have to be prepared to allow verification of several *timing constraints*:

- *Period*: As the controller behaves periodically, all necessary computations and

ITEA

adjustments of actuators must be completed within the given time interval. The period is e.g. imposed by a sensor device delivering its value periodically.

- *Deadline*: The deadline of a task is measured relative to the beginning of the period. The task has to be guaranteed to have finished before this deadline. Note that the deadline is less than the period length, if there are several tasks that have to be completed within the given period.
- *Response Time*: This is the longest time ever taken by a task from the beginning of its period until it completes its required computation, i.e. including all possible interferences by higher priority tasks and interrupt routines. The real-time constraint is represented by the fact that the worst-case response time of a task has to be smaller than its deadline.
- *Release Time*: An offset value that represents the arrival (release) of a certain task with regard to the beginning of the period. The release time of a task must not be later than (deadline - WCET), where WCET is the Worst-Case Execution Time of the task.
- *Jitter*: The amount of time the response time of a task can vary due to interferences and inaccuracies caused to it and its predecessors.



Figure 1: definition of time-constraint metrics

1.1.3 Memory Constraints

1.1.3.1 Common characteristics

Memory constraints come in two flavours: constraints on the size and on the behaviour. Size constraints are particularly important in products for mass consumer markets. Indeed, if the memory of such products has to be increased, then this will have an immediate influence on the price of the appliance. Even for products where price is not an issue, the size is best determined in such a way that ample space is left for possible future extensions. Especially when microcontrollers with on-board RAM and/or ROM are used, a substantial gain in cost can be obtained by designing the software in such a way that it fits the on-chip resources, and no additional, external memory is required.

If there is a requirement for predictable, deterministic behaviour of the memory, then this constraint has an influence on the way the memory is used. Again, there is a hard and a soft version of the constraint. The hard variety demands that all bytes that are used are somehow accounted for by the application. For the soft variety, this is not required. However, deterministic behaviour usually disallows dynamic memory use, as well as time-unpredictable features such as virtual memory, cache or garbage-collection schemes.

1.1.3.2 Generic definition

Memory constraints can be summarised as follows:

- **Cost**: minimise the size to keep it as low as possible. For microcontrollers, the cost function has a discontinuity when the size of the required memory resources exceeds what is available on-chip.
- **Load**: maximise the size to cater for future extensions
- **Hard determinism**: no dynamic behaviour, and all bytes accounted for
- **Soft determinism**: no dynamic behaviour.

1.1.3.3 Metrics

ITEA

The following metrics apply (and are quite self-explanatory):

- Size
- Load
- Cost
- Dynamic behaviour
- Accounting for the data

1.1.4 CPU constraints

1.1.4.1 Common characteristics

Two main issues are important to CPU constraints:

- The CPU performance must be well dimensioned;
- The CPU usage must be deterministic.

Performance-wise, the CPU must be sufficiently powerful so that the application's algorithms still satisfy the real-time constraints. Also, the CPU must not be used at a 100% of its capacity at this time, to allow for future expansions. If the application is intended for mass markets, then care must be taken not to go for a performance overkill solution, because that might not be cost effective.

To ensure a deterministic CPU usage, it is better to take a time-triggered approach rather than using an interrupt-driven approach, although this can not always be avoided.

1.1.4.2 Generic definition

CPU constraints can be reduced to:

- Overall performance must be sufficient to support current requirements
- Load must allow to cater for future extensions
- Determinism: no dynamic behaviour, or at least try to reduce to the strict minimum

1.1.4.3 Metrics

The following characteristics of the CPU can be measured (e.g. with a logic analyser) or estimated by code inspection and processor spec study):

- Intrinsic performance (from datasheets)
- Actual performance (to be measured with e.g. logic state analyser)
- Load (to be measured with e.g. logic state analyser)
- Determinism of the behaviour (to be checked with e.g. logic state analyser and by inspection of the code)

1.1.5 Bandwidth Constraints

1.1.5.1 Common characteristics

A number of embedded applications have very stringent bandwidth requirements, dictated by the amount of communication that is required (e.g. over buses). It specifies the maximum amount of data that can be moved over a channel, e.g. in bits per second. Furthermore, there may be constraints on the total time that a communication is allowed to take.

1.1.5.2 Generic definition

The two constraints regarding bandwidth are:

- Maximum transmission speed, which determines the maximum bandwidth the signal can/should use
- Total transmission time, which becomes important if the speed at which the data is sent, has to be artificially reduced to meet the previous constraint, either due to the amount of data or the unavailability of the full bandwidth.

1.1.5.3 Metrics

Both constraints can be measured:

- The average, maximum and minimum amount of bits, sent per second, can be monitored
- The total time of a typical, a best case and a worst-case transmission can be measured to evaluate the quality of service rendered.

1.1.6 **Power Consumption Constraints**

1.1.6.1 Common characteristics

Especially in handheld appliances, power management has an important impact on the autonomy of the system. Therefore, it must be under control right from the beginning of the design. Also, as the power is in such cases drawn from a battery, these constraints will have an impact on the overall design of the appliance, and indeed, the choice of battery. Specifically for software, care must be taken that the hardware is used only when required (e.g. if the system contains a heater or a backlight, don't switch it on when it is not necessary).

1.1.6.2 Generic definition

There are several direct and derived aspects to the power consumption constraints, and this time, not all of them can be expressed as electric or temporal quantities:

- Autonomy of the system
- Cost
- Weight
- Dimensions

1.1.6.3 Metrics

The power consumption of the embedded system can readily be measured:

- Power = voltage * current, both of which can be measured
- Energy capacity of the battery (from the battery specs)

Other quantities:

- Dimensions
- Weight
- Cost

1.2 **Functional characteristics**

Functional characteristics describe the system. Although they can be categorised, deriving metrics from them is quite impossible. This feature therefore has been omitted.

1.2.1 **Communication**

1.2.1.1 Common characteristics

Embedded systems usually have one or several means to communicate besides the User Interface. They can be classified as follows:

- Bus as an internal communication between its constituent parts
- Bus as a dedicated external communication to similar and/or different devices
- Network as an open connection to similar and/or different devices
- Physical read/write devices (magnetic/optical media, ...)

All possibilities share one feature: no matter how the connection(s) is (are) ultimately established, strict protocols are to be followed. Those protocols dictate some of the real-time and performance requirements.

ITEA

1.2.1.2 Generic definition

A distinction can be made based on the communication type:

- Internal bus
- External bus
- (Wireless) network connection.
- Carrier

They all make use of one or more:

- Protocols.

1.2.2 **User Interface**

1.2.2.1 Common characteristics

Most embedded systems perform some interaction with the human operator as well. To accomplish this, a large variety of interfaces are available:

- Keyboard
- Display with GUI
- touch screen
- voice input
- voice output
- image input
- fingerprint (for identification and authentication)
- sensors, measuring other types of data

1.2.2.2 Generic definition

A classification is needed here to describe the different I/O possibilities:

- Manual input (keyboard, touch screen)
- Visual input (scanners)
- Visual output (display, possibly with GUI)
- Voice input
- Voice output
- Sensor input

1.2.3 **Command and Control**

1.2.3.1 Common characteristics

Command and Control functions usually remain restricted to the embedded system itself, but in some applications, an embedded system may get additional authority to control other (embedded) systems of the larger system it is part of. This is usually done by allowing the embedded system to take in, or drive discrete and/or analog lines.

1.2.3.2 Generic definition

Basically, only two kinds are available:

- Discrete I/O
- Analog I/O

1.3 **Operational characteristics**

For these types of characteristics, it is oftentimes quite difficult to distil some generic definitions. Even more cumbersome (and indeed in many cases impossible) is to derive metrics from them. Therefore, definitions and/or metrics were only added where they made sense. In the other cases, they were simply omitted. Only extensive testing and verification against the defined requirements for the system can reveal whether a design goal is not met. Needless to say that in view of the large variety of systems, this task is very difficult to

generalise.

1.3.1 Quality of Service

1.3.1.1 Common characteristics

For most of the partners, this is the number one characteristic. Although “Quality of Service” is a very vague term, it can be described by the following three phrases:

- The service must be delivered with the desired functionality;
- The service must be delivered in a timely manner;
- Sometimes, also a quick readiness after switching the appliance on is very desirable.

The level of fulfilment of these aspects can vary even for one and the same embedded system, depending on its set of “missions”, or intended purposes. On one occasion, only a very basic functionality/service may be required, while put in another situation, much more is expected from the same appliance. Depending on the requirements put to it, it may happen that the additional functionality/services required from the system need not to be delivered at the same high quality of the basic functionality, and that it is perfectly acceptable that the quality goes down even further when the system is solicited even more. This is usually called “graceful degradation”.

Although “Quality of Service” is impossible to quantify, it can be broken down into other aspects, which may prove a little easier to handle:

- Functionality (see chapter 1.2): can be checked against specifications
- Timing constraints (see chapter 1.1.2): can be quantified
- Dependability (see chapter 1.3.2) for more details

1.3.2 Dependability

1.3.2.1 Common characteristics

This characteristic seems very subjective at first. Yet, it is of utmost importance for most embedded systems. The Quality of Service provided by e.g. a handheld appliance “depends” on it. In case the embedded system is used in a safety-critical situation, the safety of the operators (and possibly many more people) “depends” on it.

Operators must be able to put their trust into the system. This can only be achieved when the system has a high rate of availability (chapter 1.3.3), and provides reliable (chapter 1.3.4) service to the user. However, this is usually not enough. How can an operator put his trust in a system when there is no way of verifying the correctness of its operation? The notion of safety comes into play at this point (see chapter 1.3.5). This means that dependability can be further expressed in terms of these three characteristics, which will in turn provide a way to quantify dependability.

When redundancy is built into the system, then this will have an impact on the dependability. See chapter 1.3.4 for a discussion in combination with related characteristics.

1.3.2.2 Generic definition

Trustworthiness of a system such that reliance can justifiably be placed on the service it delivers. Availability, reliability and safety must be ensured.

1.3.3 Availability

1.3.3.1 Common characteristics

In applications where short periods of downtime are acceptable, they must be minimised in order to maximise the availability of the service that is delivered (closely related to the Quality of Service). A number of statistical methods, based on the history of the system, have proven

ITEA

their value on the hardware level. For software, however, most of the stochastic approach is still being investigated and/or developed.

Dependent on how redundancy is built into the system, there will be an impact on the availability. See chapter 1.3.4 for a discussion in combination with related characteristics.

1.3.3.2 Generic definition

Measure of correct service delivery with respect to the alternation of correct or incorrect service (dependability with respect to readiness for usage).

1.3.3.3 Metrics

Cumulative downtime or uptime over an extended period of time (possibly expressed as a percentage) are the typical measures used to describe this aspect.

Also the number of failures per (extended) period of time (e.g. a year) can provide valuable information, or even better, the mean time of occurrence of the first failure. These are statistical metrics, which can be drawn from historic data, if available.

At this time, it makes sense to distinguish between repairable systems (either subject to maintenance and/or failing systems can be replaced entirely) and non-repairable systems (inaccessible systems like satellites). For the latter ones, cumulative up or downtime has little or no meaning. If a failure causes the system to go down, then it becomes practically impossible to get it operational again.

It is not always necessary to consider the fact if a system is repairable or not, but whenever it is believed to have an impact on the characteristic, it will be taken into account.

1.3.4 Reliability

1.3.4.1 Common characteristics

The reliability is closely related to the safety: it is the length of time the system must be able to operate without the safety aspects being jeopardised. Very often, reliability and safety are therefore treated together, and usually, a trade-off will have to be made between them. A means to change reliability/safety can be adding redundant systems, or adding components to verify the correct operation of the basic functionality. The net result of redundancy is that safety increases (more error situations can be detected and trapped/reported), but that reliability decreases (the redundant system itself can fail as well).

Again, probabilistic approaches can be used to refine the reliability measure. Such a metric could be the probability that a failure occurs before a certain time elapsed. Those approaches are again much better defined for hardware than for software, and if it is not possible to derive this from historic data, an a-priori analysis is very difficult for a software application.

If redundancy is added to the system, with a decision-making algorithm to sort out the failing subsystems and (possibly) stop the whole operation as soon as the discrepancies become too big, then this will have a definite influence on the four characteristics which are closely linked together: dependability, availability, safety and reliability. In general:

- safety will go up (there are backup subsystems that can take over)
- reliability will go *down*: (there are more possible causes for failures, also the decision-making subsystem itself can fail)
- if the decision-making algorithm has the authority to shut down the overall system, then availability might go down as well (more possible failures can cause more alarming situations than before)
- the dependability will definitely be influenced by the characteristics above, but the question if it will improve or deteriorate depends on which of the three characteristics are focused.

1.3.4.2 Generic definition

Measure of continuous correct service delivery (dependability with respect to continuity of service).

1.3.4.3 Metrics

Usually, quantities like MTBF (Mean Time Between Failures) are calculated/estimated.

This metric loses its meaning for non-repairable systems. The first failure is usually fatal. Therefore, it is more important to use MTTF (Mean Time To Failure) in this case.

1.3.5 **Safety**

1.3.5.1 Common characteristics

A more stringent aspect than reliability is the safety of the embedded application. Potential hazards must be defined, and the probability of their occurrence estimated. Of course, this occurrence must be avoided at all cost, but the probability is never zero, because electronic failure rates are not zero either. In this case, possible software failures must be:

- Traced, so that they can not lead to one of the defined hazards (fault tree analysis);
- If it does lead to a defined hazard, the probability is lower than the one defined;
- Any occurrence is detected and announced to the operator, so that corrective actions are still possible.

Safety will definitely be influenced by redundancy. See chapter 1.3.4 for a discussion in combination with related characteristics.

1.3.5.2 Generic definition

Measure of continuous delivery of either correct service or incorrect service after non-catastrophic failure (dependability with respect to the non-occurrence of catastrophic failures).

1.3.5.3 Metrics

Fault analysis will lead to requirements, which need to be added to the specification requirements and they must be tested against. This metric as such is unavoidably following from a stochastic analysis.

If historic data is available, then a metric similar to MTBF can be redefined to reflect the Mean Time Between **Catastrophic** Failures, or the mean time of occurrence of the first **catastrophic** failure can be derived. For non-repairable systems, the MTTF should again be used instead.

1.3.6 **Robustness**

1.3.6.1 Common characteristics

This depends on the kind of application, but embedded systems are often used in environments where reset and/or repair are not desirable. In that case, the system must be up at all times, and it must remain functional, even when the data sent to it is erroneous. The embedded system must be able to handle any circumstance in which the overall system is used.

1.3.6.2 Generic definition

The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environment conditions.

1.3.7 **Testability**

1.3.7.1 Common characteristics

There are two distinct aspects to the testability of an embedded system: the testability of the application during development and/or qualification for use, and the ability of the embedded system to evaluate its own condition.

ITEA

Testability at development time is not an easy issue. Some formal methods/tools allow to test and verify (and in some cases: prove) that a certain automaton is behaving the way it was designed, or conversely, that it is not. A clear advantage of this approach is that a large part of the embedded software application can be tested without the need for the final hardware. The development environment can already be used. Coverage analysis is a good example of what can be accomplished in a simulated mode.

One of the topics described in the safety aspect was the ability to detect and announce the occurrence of a failure. This means that substantial efforts must be made to allow for the implementation of Built-In-Tests, which monitor the health of the system. In order to accomplish this task, it may be required that special components are added to provide the required feedback. Although the additional components will reduce the reliability, they will have a positive effect on safety. Special care must be taken when such components, or even parts of the application, are provided by a third party.

1.3.7.2 Metrics

Although it is very difficult to measure the complexity and testability of software, a number of quantities can be calculated (e.g. McCabe number). Many software development tools allow the programmer to calculate this indicator. It is to be used with care though, because it is not that difficult to influence the number by non-essential tampering with the source code.

A good indicator that can be used, however, is the amount of coverage of the requirements that can be obtained from the tests that have been conducted. Tools and formal methods exist for accomplishing this task.

Defining a metric about “how well” an embedded system is capable of self-diagnosing its current state is next to impossible at this point.

1.3.8 **Maintainability/Serviceability**

1.3.8.1 Common characteristics

For some applications, it is a must that defective systems can be replaced and/or repaired very quickly. Even when this aspect is not a requirement, it is highly desirable to ensure customer satisfaction.

1.3.8.2 Metrics

MTTR stands for Mean Time To Repair and is an important indicator for repairable systems. Although this is a very deterministic measurement, there are so many external factors that can influence the actual repair time. In case a more sophisticated measure is needed, a statistical approach may be taken, and instead of looking at the (constant) MTTR, the probability that the actual repair time exceeds the MTTR can be investigated.

In most practical situations, however, the MTTR proves to be adequate to describe a system’s maintainability.

It is clear that MTTR only applies to repairable systems.

1.3.9 **Security**

1.3.9.1 Common characteristics

Some embedded systems have direct access to public networks. The problem at hand is that the opposite also holds: the public network has access to the system. This must be handled with extreme care, and the usual solutions which hold for “standard” computers, are not powerful enough to ensure secured access.

1.3.9.2 Generic definition

Dependability with respect to the prevention of unauthorized access and/or handling of information.

1.3.10 **Field Loadable SW**

1.3.10.1 Common characteristics

Upgrading software is a very important aspect: it allows to correct possible flaws, and/or provides a possibility to enhance the functionality in the field. Care must be taken that the new software versions are still compatible with the older hardware.

1.3.10.2 Metrics

Some measures will have to be taken and verified to ensure data integrity during and after download. Needs proof by analysis.

1.3.11 **Configurable SW**

1.3.11.1 Common characteristics

From a supplier's point of view, it is very interesting to reuse large parts of previous systems. Still, different customers will have different requirements, even when the same standards are followed. Also, if the same software is to be used on different systems with different capacity, then developing the software in such a way that it is scalable is an important advantage. Control over the scale itself is then exerted through configuration. Making the software configurable is a major factor in decreasing development time.

1.3.11.2 Metrics

Care must be taken that only the parts of the software that apply to the application at hand are activated and nothing else. Needs proof by analysis.

1.3.12 **Flexible SW**

1.3.12.1 Common characteristics

Closely related to the previous topic is the flexibility of the software that is developed. Not only is it desirable to make small adjustments to a particular application (configurable), it also has to be possible to accommodate for late changes in specs, or simply reuse of a certain part of the software in an entirely different area. So, a well-thought API is always a big advantage.

1.3.12.2 Metrics

Care must be taken that only the parts of the software that apply to the application at hand are activated and nothing else. Needs proof by analysis.

1.4 **SW Architecture characteristics**

Again, generalising the following aspects of embedded systems is quite difficult. The way these characteristics are usually verified is through inspection of design/code/test cases,...

1.4.1 **Software Architecture**

1.4.1.1 Common characteristics

Embedded software is usually organised in several layers. Practice has shown that in many cases, the number of layers turns out to be three:

- I/O hardware access, operating system;
- Middleware, not really tightly linked to hardware, but providing basic services such as device drivers and interrupt handlers. Access to this layer is done through an API. A very interesting approach is to implement a "virtual machine" at this level. This way a nice path towards portability is laid down;
- Top layer contains the functionality of the application.

The main advantage of such an organisation is that general and application-dependent features can be kept apart quite nicely. Also, if needed, different software development tools can be deployed per layer.

1.4.2 Scheduling Type

1.4.2.1 Common characteristics

This is very dependent on the type of constraints that were put forward: hard or soft real-time requirements.

For hard requirements, a time-triggered approach is to be preferred, because it is a lot easier to prove that the software has a predictable behaviour.

When reaction time to external influences is of the highest priority, and hard requirements are not present, then an event-driven architecture might give better results.

A mixed approach is also possible.

For some applications, many events have to be handled in parallel, so yet another scheduling is required: concurrency. This can be achieved in close interaction with an operating system that has this functionality, or a static or dynamic scheduling scheme can be compiled directly in the application.

It will become apparent from what follows that this choice will have many consequences.

1.4.3 Interrupt handling

1.4.3.1 Common characteristics

When hard real-time requirements have the highest priority, then the use of interrupts should be avoided as much as possible, because they have a negative influence on the predictability of the software. A polling-scheme or other time-triggered approach is better.

1.4.4 Tasking and Exception Handling

1.4.4.1 Common characteristics

The same remark holds for these features, although an exception would be allowed to signal hardware failures.

1.4.5 Processor privilege usage

1.4.5.1 Common characteristics

In most cases, the mode with most privileges is used to allow direct and strict control over all hardware, which again is improving the predictability. The use of dual modes can be beneficial when strict software partitioning is required.

1.4.6 Languages used

1.4.6.1 Common characteristics

As was to be expected, a large variety of languages are used: from assembly language over C to object-oriented languages such as C++, Java and Ada. Also, modelling languages such as UML are in use, or ESTEREL for creating finite state machines. For designing GUIs, graphical rapid prototyping are used quite often.

It would seem that when safety-criticality aspects of the software have higher priority, there is a shift towards more abstraction: higher languages and modelling descriptions. Although the costs involved are usually higher, and learning curves are steeper, the long-term benefits are worth it.

1.4.7 Modularity

1.4.7.1 Common characteristics

By carefully separating the different functionality of the embedded software into different modules, the way lies open to define a consistent, recognisable template that can be reused throughout the whole application. By maintaining strict rules and discipline in the implementation of such an approach, reuse and a full-fledged component approach are within reach.

1.4.8 Portability

1.4.8.1 Common characteristics

The development environment and the language constructs used should ensure portability as much as possible, to avoid software changes each time the hardware is upgraded or replaced.

1.4.9 Configurability

1.4.9.1 Common characteristics

Whereas the previous chapter already discussed this feature to some extent, it only dealt with the static aspect of changing the configuration of the embedded software. This time, the dynamics of the program sometimes need to be changed on the fly, at runtime. If the changes control a part of the software that steers hardware, then the system may behave quite differently after such a change.

1.4.10 Operating Systems and Kernels used

1.4.10.1 Common characteristics

Again, the architecture choice impacts this aspect a lot. If requirements disallow the use of a number of features, then an appropriate choice of OS and/or kernel can make the embedded software application more deterministic, and can be a big help in proving that this is the case. Use of a stripped-down kernel, or even eliminating the need for a kernel can be a valid choice. New research efforts about component-oriented run-time layers proves very promising, but also the more "classical" real-time operating systems have their merit, depending on the type of application, because they provide extendibility towards the future.

Care must be taken when using the memory features of the processor or the operating system. For some application, it is considered no problem to make use of virtual memory and/or cache. Other applications can not allow those features to be used, again for reasons of predictability.

1.5 HW Architecture characteristics

Due to the large variety of possible used hardware components and architectures, generalisation and measurement is quite difficult, if not impossible and was therefore omitted. The characteristics of the hardware usually follow from:

- The specifications of the customer
- Decisions made in the design to meet the specifications
- Conclusions deduced from a prototype.

1.5.1 Processors

1.5.1.1 Common characteristics

As was to be expected, a large variety of processors are in use. Some trends: Intel x86, Motorola 68K, PowerPC.

1.5.2 Memory

1.5.2.1 Common characteristics

ITEA

Various combinations of ROM, RAM, Flash and EEPROM.

1.5.3 Internal Buses

1.5.3.1 Common characteristics

PCI (PC –platform), compact PCI

1.5.4 Communication Buses

1.5.4.1 Common characteristics

Some industry standards like CAN are clearly present, and domain specific buses are there as well.

1.5.5 Other Outside Connections

1.5.5.1 Common characteristics

Mainly analog/digital I/O directly, and also A/D D/A.

1.6 Development Tools

1.6.1.1 Common characteristics

Various commercial and also experimental environments to control several aspects:

- High-level design tools + code generators
- Language-specific development environments
- Graphical rapid prototyping + code generators
- Configuration management tools
- Test case generation packages
- Documentation generation packages
- Scheduling tools
- Problem reporting/tracking tools
- Requirements tracking tools.