By Alan Moore,
Vice President of Product Strategy,
ARTiSAN Software Tools, Inc.

Using UML to Design an Embedded System Introduction

INTRODUCTION

An embedded system is a complex affair involving many different layers; at the bottom some kind of hardware, at the top some application functionality (often in software) and in between various types of infrastructure, such as operating systems and protocol stacks. This article discusses one approach to describing the design for such a system, using a set of layers. The approach is broadly supported by the current version of UML (1.4), but some suggestions are given for improvements, most of which are certainly within the scope of UML 2.0.

LAYERED DESIGN

This particular approach leads to a multi-layer model of the system design, with each layer corresponding to a different layer of abstraction; this will be familiar to engineers in the telecom domain, where the seven layer ISO model has had a very powerful influence. The content of layers is described using standard UML concepts, although in the following example some useful stereotypes (the UML mechanism used to support extensions) are introduced. Mapping between layers is accomplished using a UML relationship called "realization", a stereotype of the abstraction relationship, which relates a concept in a higher level of abstraction (sometimes called a specification) to one at a lower level of abstraction (sometimes called a realization). This approach, of regarding different levels as views of the system at different levels of abstraction is a powerful, though only narrowly used, capability of UML.

EXAMPLE

The example is a generic 'instrument', which gathers data from sensors and displays the data on a screen. This article discusses a simple layered design for the system, from application functionality through to hardware. The three layers in this design are as follows:

- **Application** - the layer that solves the application problem.
- **Hardware** - the layer that describes the physical environment.
- **Infrastructure** - the layer that implements the application on the hardware with adequate quality of service.

APPLICATION DESCRIPTION

This layer describes the subject matter of the problem that the application is going to solve, in this case the capture, processing and display of sensor data.

Figure 1 is a UML collaboration diagram, which shows the structural layout of the application. The various objects are responsible for different aspects of the application. The classes of these objects, i.e. the description of their structure and behaviour, are not shown on this diagram; as often happens in embedded systems, many of the classes have only a single object instance.

The three major objects are Data Capture (of class cCapture), which takes the Raw Data (class InstrumentData) from the sensor; Data Filter (of class cFilter), which applies some kind of filtering to the Raw Data; and Data Display (class cDisplay), which displays the Filtered Data (also of class Instrumentdata). There is also an object, Instrument Calibration, in charge of helping the user to calibrate the instrument.

The objects in this diagram, with their associated class definitions, describe the whole system at the application level of abstraction. However, this is by no means the end of the story; the next section looks at the hardware on which this application needs to run.

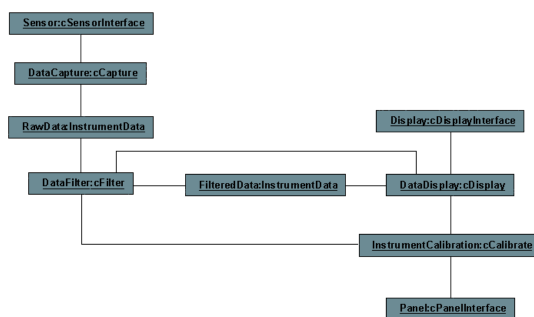

Figure 1. Collaboration of the objects in the application layer.

HARDWARE DESCRIPTION

The hardware layer in this example describes entities in the physical environment of the system, such as processors, boards and buses; UML, through its notion of deployment, is suitable for describing this level of hardware detail. Although arguably it is possible to use UML to describe the hardware architecture below this level of abstraction (for example by using classes to represent hardware blocks), this is not generally the case and is not very effective. Languages like VHDL and Verilog, and the tools that use them, are a better choice for more detailed hardware design.

Figure 2 shows an extended variant of the UML deployment diagram with various different node stereotypes, some showing interfaces as symbols on the edge of a node symbol, rather than the traditional 'lollipop' used in the description of class and component interfaces; this symbology seems to be more commonly understood in the hardware world.

This is a fairly traditional design, with a DSP board and

Copyright 2001 by Dedicated Systems Magazine - 2001 Q2 (http://www.dedicated-systems.com)

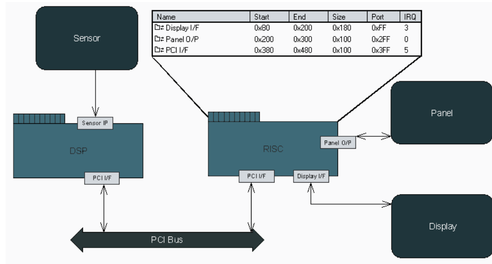

Figure 2. The physical hardware layer.

a RISC board connected through a PCI bus. The details of the RISC board's interfaces are shown; capturing this information can be very useful for engineers working on the more abstract layers. As with the application layer, this diagram describes the whole system, but from a particular point of view.

INFRASTRUCTURE DESCRIPTION

This layer describes the infrastructure (operating system, communications, middleware etc.); this infrastructure is not part of the application functionality, nor is it the hardware, but is required to "realize" the functionality on the physical environment. The composition of this kind of description will depend heavily on the physical hardware and the resources available. It will range from a simple micro-kernel with a main thread and interrupts, to a complex n-layer distributed middleware solution.

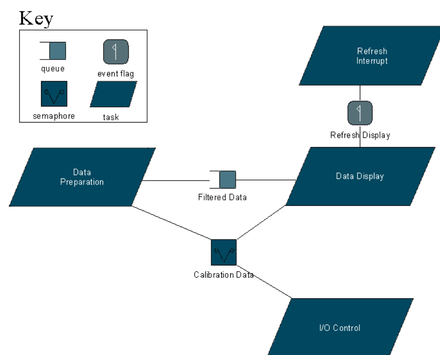

Figure 3. The infrastructure layer.

Figure 3 shows the 'object's in the infrastructure layer, using a variant (the UML term is "stereotype") of the collaboration diagram, where standard types of operating system feature (task, mutex etc.) are shown using special icons.

Three tasks are used:

- **Data Preparation** - to sense and filter the incoming data.
- **Data Display** - to display the filtered data.
- **I/O Control** - to perform calibration and manage user settings.

There is also one interrupt from the Display device that requests a display refresh, which is handled by Refresh Interrupt. Various operating system primitives are used for communication between the tasks: the Calibration Data semaphore provides mutual exclusion on calibration information, the Refresh Display event flag provides an asynchronous mechanism for the interrupt handler to trigger a refresh; the Filtered Data queue allows filtered data to be communicated from Data Preparation to Data Display.

This diagram, like the previous two, describes the whole system but from a specific point of view, the concurrent entities and the communication between them.

So far, for the purposes of this article, the three layers in the design have been shown in isolation, although during the design process all three would be considered together. In the final section of the article, the mapping between the layers is discussed in detail.

MAPPING BETWEEN DESCRIPTIONS

The mechanism for mapping between the layers is based on creating a "realizes" relationship between the elements in the layers being mapped. Figures 4 and 5 show two alternative visualizations, neither of which are official UML views. UML does not cover the use of the "realization" relationship in much detail, but it is an important focus both for the "RT Profile" (profile for Timeliness and Schedulability) of UML, which is approaching its final submission, and for UML 2.0.

Figure 4 shows one diagrammatic way of representing the realization between layers. Objects in the layers are connected using "realizes" relationships of various types (the use of guillemets ("") is a standard way in UML of denoting a stereotype). The "realizes" relationships are intended to show the mapping between concepts in the application and those in the infrastructure; this is not just limited to objects, for example the link between the Data Filter and Data Display objects is realized by the Filtered Data queue. Nor is there always a one-to-one mapping; the Data Preparation and Data Display tasks both realize a "copy" of Filtered Data, the reason for which is revealed in Figure 5; "copy" is a specialized stereotype of realization that has a specific purpose.

Some of the objects in Figure 4 are grouped to reduce the number of relationships between the layers; this is just for convenience in this illustration, but does raise an important issue of handling complex, detailed real-

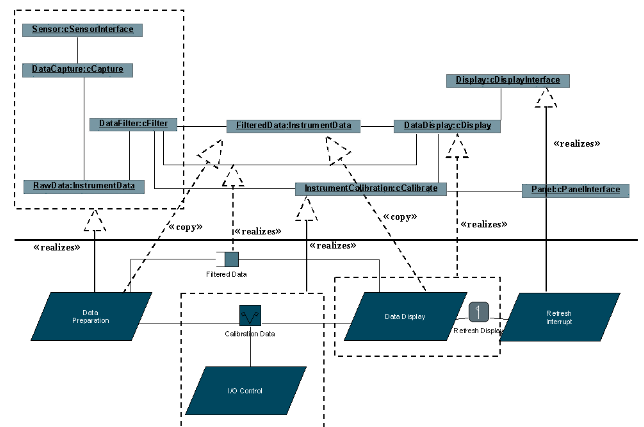

Figure 4. Mapping application to infrastructure.

Copyright 2001 by Dedicated Systems Magazine - 2001 Q2 (http://www.dedicated-systems.com)

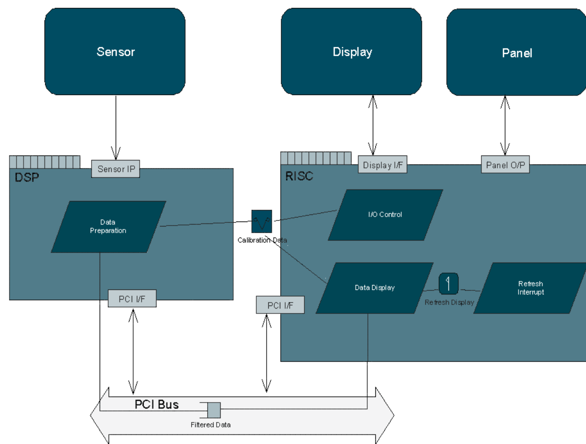

Figure 5. Mapping infrastructure to hardware.

izations, something that UML 2.0 will have to deal with.

Although these realizations have meaning, the actual implementation may still require a lot of hard work to achieve. The classes in the design specification have to be able to support these realizations, and engineers will have to decide whether to use traditional API calls, or some form of code generation, to implement the relationships. Nonetheless these realization relationships effectively express the intent of the designer, which is of utmost importance.

Figure 5 shows another diagrammatic representation that is plausible in the UML 2.0 timeframe. The lower level (realization) is shown behind the higher level (specification), implying the ”realizes” relationships.

This diagram shows which tasks sit on which boards, and shows the realization of the queue between Data Preparation and Data Display. The reason for two copies of Filtered Data in the infrastructure (one for Data Preparation and the other for Data Display), as shown in Figure 4, now becomes clear; it is because the tasks run on separate processors with no shared memory. The diagram also reveals that we have a problem with the ’shared’ calibration data - the resolution of this issue is left as an exercise for the reader!

This is a traditional embedded design; in a System-on-Chip implementation, the IP for the RISC and DSP processors, plus all of the infrastructure and application functionality might be realized in turn on an ASIC, thus adding another layer. As with the previous diagram the implied implementations may require significant effort. In the ’traditional’ realization described in the example, the application and infrastructure layers have to be coded, compiled and linked for the RISC and DSP chips. In the case of an ASIC, detailed synthesis is probably required in addition. However, the instructions to the implementer are clear, which is a very important consideration.

Either diagram type could be used to show either mapping, and each diagram style has its different strengths and weaknesses. In Figure 4, it is easier to show the detail of realizations, whereas in Figure 5, the detail is harder to show but the overall structure is clearer; both types of diagrams quickly become cluttered. Ideally one would need 3D diagrams with perspective, which would allow the layers to be clearly seen and yet allow realization relationships to be drawn between them. However, nobody is proposing (yet) that UML 2.0 become a 3D graphical language, thus restricting the available options to 2D flattened views like that in Figure 4 and Figure 5.

CONCLUSION

A layered approach to the description of embedded systems has the benefit of providing clear abstractions at different levels of the design, with the ability to show the mapping between the levels, thus guiding the implementers. The current version of UML (1.4) has many of the required concepts to support this, such as:

- A stereotyping mechanism to allow different abstraction layers to use appropriate symbology.
- A ”realizes” relationship to show mappings between layers.

However, the use of the ”realizes” relationship is poorly explored in the UML standard and in particular the visualization of realization needs more attention moving forward. Fortunately both submitters of the profile for Timeliness and Schedulability [1], and the groups working on the UML 2.0 superstructure [2] are addressing aspects of this problem, so I do expect this approach to be more clearly and powerfully supported in the future ■

Alan has 15 years of experience in the development of real-time and object-oriented methodologies, and their applications in a variety of problem domains. He has been actively involved in product development, training and consulting related to OOAD and structured development tools during that time. Alan has co-authored a book on GUI design and published several papers, and has lectured on a wide variety of analysis and design issues.
Alan is responsible for the specification, planning and management of the ARTiSAN product strategy. He is the author of ARTiSAN Real-time Perspective, a pragmatic approach to the development of real-time systems and is an active participant in the Real-time Analysis and Design Group (RTAD) of the Object Management Group (OMG).

REFERENCES

1. UML Profile for Scheduling, Performance, and Time RFP (OMG document ref: ad/99-03-13)
2. UML 2.0 Superstructure RFP (OMG document ref: ad/2000-09-02)