

7 *Implementation of CIP Models*

This chapter is not yet complete. It already contains the description of how a CIP model is implemented. The chapter will be completed by executable implementation examples.

7.1 *The CIP Model Implementation Task*

A CIP system describes the behaviour of an embedded system by means of an executable behaviour model. This model is abstractly connected to the external processes by a set of message channels. The CIP model implementation task is therefore twofold. First, the behavioural model, represented by a collection of composed extended finite state machines (clusters), must be transformed into executable target code. This task is fully automated by the code generators of CIP Tool. Second, the channel connections between behavioural model and the external processes must be implemented by means of peripheral devices and corresponding interface modules. The corresponding implementation task can in general not be automated because it highly depends on the used interface technology. In addition, if hard real-time conditions have to be satisfied, the connection implementation includes domain-oriented scheduling of concurrently occurring events.

7.1.1 *Functional and Connection Layer*

The abstraction supported by a CIP model is based on the domain-oriented decomposition of the embedded system problem into a *functional* and a *con-*

7 Implementation of CIP Models

nection problem, as described in chapter 3. The resulting layered system architecture of the final system is depicted again in Figure 48.

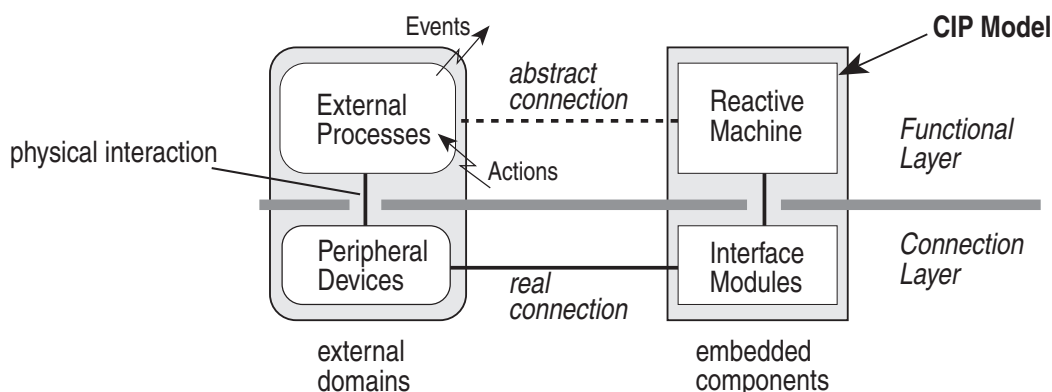


Figure 48: Layered embedded system architecture

The *External Processes* domain represents the chief problem domain. It includes the physical processes to be controlled, system operators that command and supervise the system, and possibly some already existing software components that must collaborate with the embedded system to be built. The *Peripheral Devices* domain represents the sensors and actuators as well as standard I/O devices of the man-machine interface.

The functional problem of monitoring and controlling the external processes is solved by building a *Reactive Machine*, modelled by a collection of cooperating CIP clusters. The *Reactive Machine* communicates abstractly via CIP channels with the *External Processes*. The connection problem consists of building a real connection that implements the abstract connection by means of *Peripheral Devices* and *Interface Modules*.

7.1.2 Implementing a CIP Model

A CIP system models the *Reactive Machine* and its abstract connection to the *External Processes*. The *Reactive Machine* is modelled by a set of concurrent clusters, each one composed of a number of extended state machines termed processes. The abstract connection is modelled by means of a number of source and sink channels that transmit messages from and to the *External Processes*.

Thus, the task of implementing a CIP model is twofold. First, the cluster model of the reactive machine has to be transformed into software components that can run on the target system. Because CIP clusters are executable behavioural models, their implementation can be fully automated by means of suitable code generators. If all clusters are implemented on the same processor, the

code for the *internal channels* of the CIP model can also be generated. This shows that the functional problem of constructing cooperating control components is solved completely by the CIP model. However, if a CIP model is to be implemented on more than one processor, the implementation of those internal channels which connect clusters running on different processors cannot be generically automatized. The implementation of channel connections between distributed clusters must obviously be based on the particular communication technology used to connect the distributed processors.

Second, the abstract connection modelled by source and sink channels specifies the real connection between external processes and generated control components. Constructing this connection is the hard part of the CIP model implementation. The real connection consist of hardware and software parts, termed in Figure 48 *Peripheral Devices* and *Interface Modules*. The construction and installation of the peripheral devices is beyond the task of the software engineer. We therefore suppose that the hardware connection already exists. What remains to be done is building the interface modules. This task may be difficult and highly non-trivial due to the rich and heterogeneous set of sensors, actuators, operator and visualisation devices that have to be connected with the generated control components. Developing the implementation of the specified abstract connection between external processes and reactive machine represents in fact the solution task of the connection problem, initially separated from the problem of building an embedded system.

7.2 How a CIP Model is Implemented

A CIP model implementation consists of transforming CIP clusters into an executable and possibly distributed reactive machine, and of developing interface modules which connect the generated code to the peripheral devices. Due to the explicitly specified interface between the behavioural model the abstract connection, the CIP model and the implementation of it's abstract connection can be developed concurrently.

7.2.1 Implementation of the Reactive Machine

The clusters of a CIP system represent concurrent reactive components that could be implemented each on a single microprocessor. However, the number of microprocessors of the target system is usually smaller than the number of clusters of a CIP model. The reason is that clusters are not only used to model

7 Implementation of CIP Models

the distributed nature of the target system, but also to decompose the system into parallel modular entities. In fact, the idea of conceptual concurrency is applied in every embedded system that is implemented by means of multitasking. Tasks are usually designed as functional threads that perfectly separate sequential control behaviour.

The obvious idea of implementing each cluster by a task that is scheduled by a real-time operating system is a possible implementation concept. The implementation of channel communication between clusters that run on the same processor could then be based on synchronisation primitives provided by the real-time kernel. However, a main drawback of the multitasking implementation concept is that the scheduling of event messages must be based on task priorities. We thus often encounter the difficulty of fitting the rich structure of desired event message priorities to the simpler structure of task priorities.

CIP therefore proposes a simple implementation structure that only matches the distributed nature of the target system. In order to specify how clusters are allocated to the various processors of the target system, a CIP model is partitioned into a corresponding number of *cluster groups*. Each cluster groups represents a subsystem that is transformed automatically into an executable *CIP Unit*. CIP units are passive software components that must be driven by interface modules responsible for event detection and message scheduling. In addition, each interface module must provide a number of action message functions that access the peripheral device output interface.

In particular cases, low level priorities can be associated to subsystems running on the same processor. In such cases the multitasking and the CIP Unit implementation concept can be combined. For instance, suitably formed low priority CIP units can be implemented together with their interface modules as low priority tasks. These tasks are enabled when no events have been detected by the interface module of the high priority CIP unit.

Specifying CIP Units

The reactive machine specified by a CIP model can be automatically implemented by a number of concurrently executable CIP Units. Each CIP Unit is a generated reactive software component that implements a number of clusters of the CIP model. Figure 49 below shows an example of a CIP model that has been partitioned into two cluster groups to specify an implementation that consists of two CIP Units.

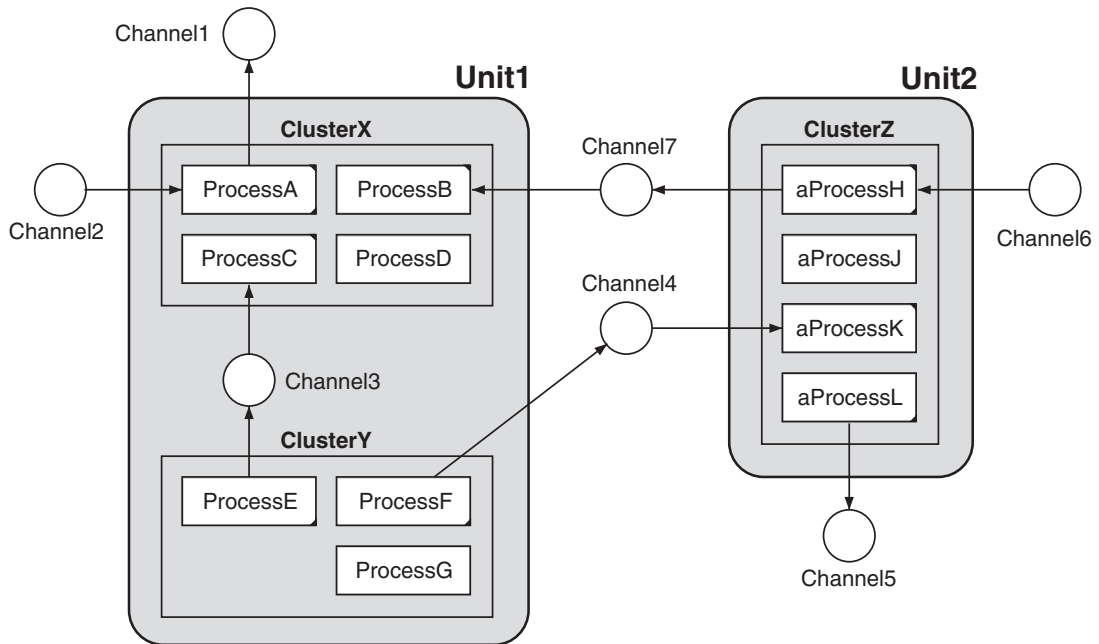


Figure 49: Partition of a CIP model into two cluster groups

Each particular partition of a CIP model into cluster groups leads to a corresponding classification of channels into *global* and *local* channels:

The set of *global channels* associated to a CIP Unit consists of all channels that leave and enter the cluster group. These are either source and sink channel that connect to the system environment (c. f. *Channel1* and *Channel2* in Figure 49), or channels that connect to processes of another cluster group (c. f. *Channel4*). The implementation of global channels must be based on the particular target interface technology, thus no generic code generation is possible.

The set of *local channels* of a cluster group consists of those channels that connect processes contained in that cluster group (c. f. *Channel3*). The implementation of local channels can optionally be generated by CIP Tool.

CIP Unit = CIP Shell + CIP Machine

The code of a CIP Unit consists of two parts: a CIP Machine and a CIP Shell.

The *CIP Machine* is the reactive behavioural body of the CIP Unit and is specified by a group of clusters.

The *CIP Shell* represents the interface of the CIP Unit and is specified by the set of channels that enter and leave the corresponding cluster group.

7 Implementation of CIP Models

The CIP Shell and CIP Machine generation is accomplished in two different generation steps. Before generating the CIP Machine code, the tool checks the CIP Machine against the CIP Shell. All global channels associated to a cluster group must appear in the corresponding CIP Shell specification. However, a CIP Shell may also contain channels that are not connected to any process. The code of the generated CIP Machine is then automatically extended by corresponding input error functions which are invoked when a message of such a channel is sent to the CIP Unit. Specifying a CIP Shell that contains more channels than necessary can be useful in certain intermediate development cycles. You can, for instance, test the code of an incomplete CIP model by linking it to a CIP Shell that is already connected to all external devices.

The possibility of specifying the CIP Shell and the CIP Machine independently crucially benefits the underlying development process. When we start developing a CIP model we ought to begin by defining the model interface, that is, we create a suitable set of source and sink channels. From the specified set of channels we can already generate the CIP Shell at an early stage of the of our modelling activities. This is even possible before creating any processes in the CIP model. A generated CIP Shell can, for instance, be used to test early channel implementations by attaching dummy CIP machines. The same procedure can be repeated during the whole software life cycle: newly required system extensions can be developed by firstly defining the interface extension, and secondly by extending the behavioural body of the concerned CIP Units.

The C Code of a CIP Unit

A CIP Unit is a *passive* reactive software component. Passive means that the CIP Unit does not have an own control thread. If C has been chosen as implementation language (the Java implementation is described below), the code of a CIP Machine consists simply of a collection of C functions, each implementing a process of the corresponding cluster group. The static variables of these process function hold the current state vectors of the implemented processes. Pulse cast interaction between processes is implemented by corresponding process function invocation. State inspection and mode control is based on corresponding shared variable dependencies.

The CIP machine is activated by function calls when its interface module has detected an event or a message sent by an other CIP Unit. The externally invoked function of the CIP Machine returns only when the activated cluster transition has run to its end. Thus, the implementation of run-to-completion semantics for an activated cluster is, in fact, simply based on locally nested function invocations.

The CIP Shell is implemented by a linear structure (`struct`) of function pointers, one for each input message and one for each output message of the CIP Shell channels. All function pointers of the CIP Shell are initialised during the CIP machine initialisation cycle. The input message functions of the CIP Machine are linked to the CIP Shell by the generated CIP Machine itself while the initial linking of the output message functions must be performed by the interface module that implements these functions.

7.2.2 Implementation of Interface Modules

The interface modules represent the active parts of the embedded software. This corresponds to the nature of CIP models, describing passive state machine compositions that are connected to each other and to the environment by means of active message channels. Remember the channel semantics which requires that every sent valid message must be consumed by the connected reader process.

Figure 50 depicts the architecture of a simple implementation of a CIP Unit and its connection to the peripheral devices interfaces and communication ports.

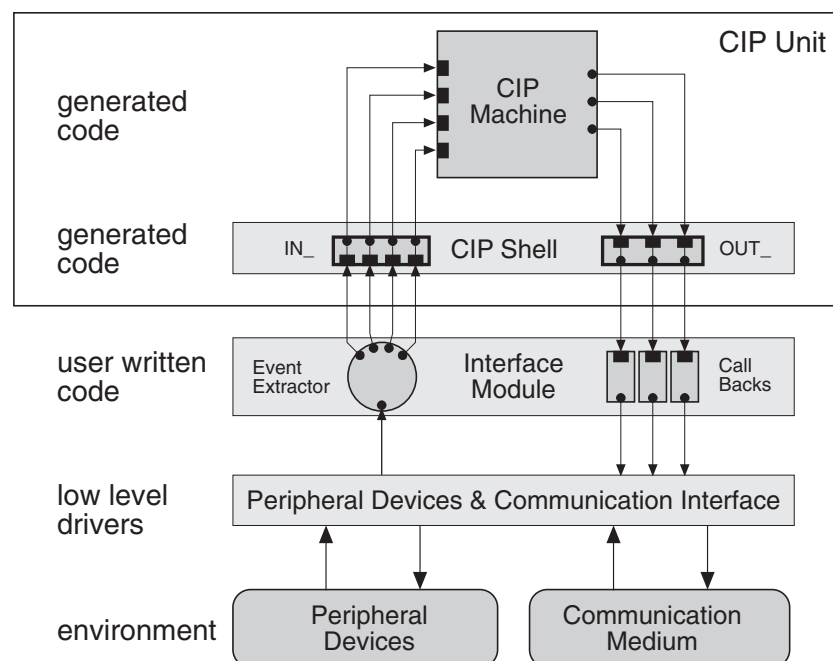


Figure 50: Implementation architecture of a CIP Unit running in a main program

An interface module consists of an active part (*Event Extractor*) that handles external input signals, and of a passive part (*Call Backs*) that creates external

7 Implementation of CIP Models

output signals. The active input part is typically realised as cyclic main program that polls sensor interfaces as well as communication port flags, and that triggers the CIP Unit correspondingly when an occurred event or external message has been detected. The output part of an interface module consists always of a number output message functions. Each of these functions must cause a particular external action through the actuator or communication output interface.

The *Event Extractor* in Figure 50 represents an active main loop that repeatedly scans the *Peripheral Device & Communication Interface*. At the *Communication Interface* appear CIP messages that are sent by an other CIP Unit of the distributed CIP model implementation. In general, an event extractor may consist of several nested polling cycles, interleaved with time periodic sampling loops. In addition, an important task of event extraction can often consist of invoking suitable data conversion routines.

When an event or a sent CIP message is detected, the CIP Machine is triggered by invoking the corresponding input message function via the input part of the CIP Shell. The CIP Machine reacts by executing the triggered cluster transition which may cause invoking some output message functions via the output part of the CIP Shell. When the activated cluster has run to completion, the invoked input message function returns control to the *Event Extractor*.

Events may also be detected by interrupts. The event identifier and associated event data have then to be stored in a event message queues. The queue is triggered for message release by the event extractor when the reactive machine is not active. In certain cases even output message functions cannot directly access the peripheral device interface and must store the corresponding action messages in an output queue.

Scheduling

In general, a scheduling problem arises as soon as concurrent messages have to be received in a sequential order. As CIP channels specify concurrent message passing connections that are implemented on a small number of processors, message passing scheduling is usually required on several levels.

First, the CIP modelling framework claims run-to-completion semantics of clusters. Thus we must sequentialize the occurring input messages for each cluster.

Second, because the interface modules usually run on the same processor as the CIP clusters, we must decide when sensors are polled, and when extracted events messages are transmitted to the clusters. In addition, sensor variables represent parallel input signals, thus the polling order to be chosen represents a

main scheduling concern. Similarly, we must often decide in which order actuators are set and cleared. In addition, a complex action may even require an own process to cause a particular external action.

Third, if clusters are implemented within different tasks running on the same processor, we have to define task priorities to tune the scheduling services of the real-time kernel.

The scheduling strategy to be chosen depends crucially on the required real time conditions and on the hybrid character of the system to be built. A main benefit of modelling reactive behaviour by means of CIP models is that the reaction time of each cluster is bounded, and that the *worst case reaction time* for every input message can be determined by measurement. Scheduling can thus be based on simple static scheduling algorithms. The proposed concept allows verifying that real-time dead lines are always met by the implemented system. In addition, if quasi continuous control functions are integrated in the CIP model, we can define event extractors that react on sampling events with zero delay, thus preventing any jitter when the corresponding control functions are executed.