# AN ASYNCHRONOUS CONTROLLER FOR A DAISY-CHAINABLE VME BUS INTERRUPTER

**Jochen Beister, Matthias Kuhn, Ralf Wollowski**

Department of Electrical Engineering
University of Kaiserslautern, D-67653 Kaiserslautern, Germany

***Abstract***

*An interrupter for use in a daisy-chained VME bus interrupt system has been designed and implemented as an asynchronous sequential circuit. The concurrency of the processes posed a design problem that was solved by means of a systematic design procedure that uses Petri nets for specifying system and interrupter behaviour, and for deriving a primitive flow table. Classical design and additional measures to cope with non-fundamental mode operation yielded a coded state-machine representation. This was implemented on a GAL 22V10, chosen for its hazard-preventing structure and for rapid prototyping in student laboratories.*

## INTRODUCTION

Teaching interrupt and priorization techniques is one of the aims of our microcomputer laboratory. We use an M68000-based VME bus interrupt system (Figure 1) that consists of a single handler and several peripheral units on the daisy chain of their interrupters. The peripheral devices are either the four end switches of a solar cell panel with limited rotation about a horizontal and a vertical axis, or - for preliminary experiments - special circuitry allowing concurrent interrupt requests by hand (Kuhn 1991). Interrupts can be requested on seven priority levels via the request lines $\overline{\text{IRQ1}}$ to $\overline{\text{IRQ7}}$. If concurrent requests are on different levels, the unit with the higher level takes precedence regardless of its position in the daisy chain, but requests on the same level, e.g. level 2 for units 1 and 2 in Figure 1, are priorized according to position.

The processes in the system feature concurrency, conflicts that require arbitration, and more or less implicit timing conditions. Asynchronous circuits are a natural solution for controlling such processes, and also offer high speed and low power consumption. But they are notoriously difficult to design, especially since concurrent processes lead to non-fundamental mode operation. Much of our research effort, however, has gone into a systematic procedure and tools for designing assemblages of communicating asynchronous sequential circuits for the distributed control of concurrent processes from a Petri net specification of required behaviour (Beister and Wollowski 1993). Therefore we rejected an artificially clocked synchronous solution for the interrupters in favour of an asynchronous design.

## BEHAVIOURAL SPECIFICATION

An interrupter in the system shown in Figure 1 interfaces with an environment consisting of its own peripheral unit, the interrupt handler (via the bus), and the neighbouring interrupters in the daisy chain. Its only activities are interactions with this environment. Hence it would be very difficult and error-prone to try to design an isolated interrupter without sufficiently precise knowledge of the overall system processes. The first design step, therefore, was to construct a high-level Petri net description of required system behaviour. This net is shown in Figure 2. The transitions represent actions of system components, and the participating signals across the *interface to and from the interrupters* are listed in the transition boxes. Some explanations follow:

- An interrupt request is modelled by letting one of the transitions labeled "Interrupter i requests…" put a token on place P in the handler section of Figure 2. The number of tokens on P is the number of pending requests.
- The peripheral units we use (but not their interrupters) withdraw their request signal $\overline{\text{IRQPi}}$ only upon being serviced. Figure 2 models the necessary - but not sufficient - condition, namely that the interrupt service routine has been initiated.
- An interrupter has to decide whether to absorb an incoming acknowledgement ($\overline{\text{IACKINi}}$) or to pass it on to the next in line ($\overline{\text{IACKOUTi}}$). It will pass it through either if it has not requested an interrupt (Pi empty, ti´ enabled, ti´´ not enabled), or if it has issued a request (token on Pi, ti´ inhibited, ti´´ enabled) but the acknowledged priority on A1 - A3 does not match its own.

The interrupters in the system are all of the same type. To simplify the design, we partitioned the interrupter into data path and controller as shown in Figure 3. Only the data path interfaces with the bus. It serves to personalize the interrupter: By means of a jumper, the request signal IRQC issued by the controller is switched to one of the seven VME bus lines $\overline{\text{IRQ1}}$ to $\overline{\text{IRQ7}}$; and by setting three switches in the "choice…" block, one of eight previously defined status/ID bytes can be selected. The data path also compares the acknowledged priority level on A1 - A3 with its own request priority as selected by the jumper, and sends a match/no match signal M to the controller. Since personalization is done in the data path, the controllers are exactly alike in all interrupters.

The next step was to derive a complete signal-level specification of the required processes observable at *the interface of the controllers and their common environment.* The result, shown in Figure 4, is stated in terms of causal dependencies between binary interface signal transitions (edges) and takes the form of a labeled and interpreted Petri net: a signal transition Petri net. This type of net, first proposed by Wendt (1977) for the present purpose, is similar to the signal transition graph (STG) of Chu (1987), but is much less restricted than the STG concept. Each transition is labeled with the leading or trailing edge of a binary signal, and the firing of a transition represents an occurence of this particular edge. Note that Figure 4 does not contain any references to internal states or state changes: the introduction of internal states is a matter of design, not of specification. The only exception is the unlabeled, shaded transition in the middle of each interrupter section. It signifies an irrevocable decision to be made by the controller regardless of its implementation, but not observable at its output. Some important features:

- A controller´s decision to absorb or pass through an acknowledgement must now be modeled as a three-way arbitration of the race between IRQPi↑ and $\overline{\text{IACKINi}}\downarrow$ ("IRQPi↑ wins", "$\overline{\text{IACKINi}}\downarrow$ wins", and "tie"). The need for arbitration occurs when the acknowledgement of a request made a number of machine cycles ago from further down the daisy chain is intercepted by a new request. The tie (a matter of nanoseconds of delay between the racing edges) is now decided (fairly!) in favour of passing through. In this minor detail, the more precise signal-level specification now differs from the high-level one of Figure 2.
- A timing condition hidden in Figure 2, but necessary for correct operation and always fullfilled in the real system, is now explicitly stated in Figure 4: The $\overline{\text{IACK}}\uparrow$ signal from the handler will always have propagated down the daisy chain and been absorbed at its true destination *before* the handler begins to process another pending request. This is modeled by the arcs with the additional dots from the $\overline{\text{IACKOUTi}}\uparrow$ / $\overline{\text{IACKIN}}$(i+1)↑ transitions near the bottom of Figure 4 to the pre place of the "compare priorities" transition of the handler.
- The matching of acknowledged and requested priority, performed by the data path, is now reflected on the signal level by the changes of controller input Mi (transitions Mi↑ and Mi↓ in the interrupter sections of Figure 4).

In a final specification step, the Petri net of Figure 4 was *formally* decomposed by extraction (Beister and Wollowski 1993), yielding the signal transition Petri net specification of a single controller´s interaction with its environment (Figure 5). A number of dependencies that are causal in the overall system now appear as timing constraints (dotted arcs) from the point of view of the individual controller. It also becomes apparent that the controller must regard $\overline{\text{IACKIN}}\uparrow$ and $\overline{\text{DSO}}\downarrow$ as concurrent even though the handler generates $\overline{\text{IACKIN}}\uparrow$ before $\overline{\text{DSO}}\downarrow$. But since $\overline{\text{DSO}}$ is fed directly to all controllers while $\overline{\text{IACK}}$ must propagate along the daisy chain, a controller may see their changes in any temporal sequence depending on its position in the chain.

Figure 5 is a precise and complete specification of the interface behaviour required of a controller communicating with its environment, and the formal basis for designing the controller.

## STATE MACHINE DESIGN

It is now possible to derive a primitive flow table for the controller as an asynchronous circuit. The construction is *algorithmic* (Beister and Wollowski 1993), and tools for performing it have been created (Simeth 1990). Essentially, the Petri net of Figure 5 is put through its evolutions by playing the token game, and the primitive flow table is generated by letting the primitive state change with the markings. The primitive states and their number - 52 for the controller - are thus found automatically, and the flow table returns to the same primitive state only if the net returns to the same marking.

At this stage of the design, concurrent input changes that always imply non-fundamental mode operation were treated first as single and then as multiple-input changes in fundamental mode. The primitive flow table was then subjected to classical state reduction, resulting in a fully reduced flow table of only 10 states, named A to K in the state diagramm of Figure 6, and with MOC and MTT behaviour. Note that the arbitration discussed above is implicitly contained in the flow table, and that we do not use a separate arbiter in the implementation.

A state assignment designed to avoid critical races and also to cope with the hazards of non-fundamental mode (nFM) operation was now constructed using only four state variables. This required the flow table to be expanded by the states L, M, and N. State transitions that had to be added in order to deal with nFM operation have been drawn as broken-line arcs in Figure 6.

## GAL IMPLEMENTATION

An FPGA or standard cell implementation being beyond our financial reach, we decided on a GAL implementation. It is a cheap yet compact solution that offers an essentially two-level structure for the combinational circuits, wide fan-in of the gates, a direct feedback option, short delays and small differences of path delays: features favourable for dealing with hazard problems. The next-state and output circuits were designed in complete sum form to avoid logic hazards. Because of the large number of product terms in the complete sums we had to choose a GAL 22V10. Four of its output logic macro cells (OLMCs) driven in combinational mode were used for the outputs, and four more in direct feedback mode for the next-state circuit. The remaining two OLMCs could fortunately be used for an asynchronous JK flip-flop needed as part of every peripheral unit for generating the IRQP signal. This raised the overall utilization of the GAL to 53%.

The circuits were built and tested directly in the VME bus system and found to operate correctly - both singly and in a daisy chain. Seven interrupters have been operating reliably for over two years under heavy workloads in our student microcomputer laboratory.

## CONCLUSIONS

We hope to have shown that  assemblages of asynchronous circuits can be adequately implemented on FPL devices, and that the use of FPLs for this purpose requires a systematic high-level design procedure if it is to be efficient. Furthermore, correctness by construction due to a systematic and - where possible - algorithmic procedure justifies the laborious task of setting up the initial overall Petri net specifications (Figures 2 and 4 in the present example) and extracting the circuit´s signal transition Petri net (Figure 5) by decomposition. Finally, FPL implementation as a means of rapid prototyping makes it possible for students to experiment with and apply asynchronous circuits in laboratory work and projects.

## REFERENCES

Beister, J., and Wollowski, R., "Controller Implementation by Communicating Asynchronous Sequential Circuits Generated from a Petri Net Specification of Required Behavior", in *Synthesis for Control Dominated Circuits,* G. Saucier and J. Trilhe, Eds., Amsterdam: Elsevier, pp. 103 - 115, 1993.

Chu, T. A., Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications", *Proc. of the Int. Conf. on Computer Design*, pp. 20 ff., 1987

Kuhn, M., "Verbesserung des mathematischen Modells einer Nachführeinrichtung und Entwicklung einer Versuchsreihe zur mikrorechnergesteuerten Nachführung eines Solarzellenpaneels", Diploma thesis, Dept. of Elec. Engineering, University of Kaiserslautern, Germany, 1991.

Simeth, M., "Ein Programm zur Konstruktion des Elemetarautomaten aus der ¸Kommunikationsregel eines asynchronen Schaltwerkes", Diploma thesis, Dept. of Elec. Engineering, University of Kaiserslautern, Germany, 1990

Wendt, S., "Using Petri Nets in the Design Process for Interacting Asynchronous Sequential Circuits", *Proc. of the IFAC Symposium on Discrete Systems,* Dresden, GDR, vol. 2, pp. 130 - 138, 1977.

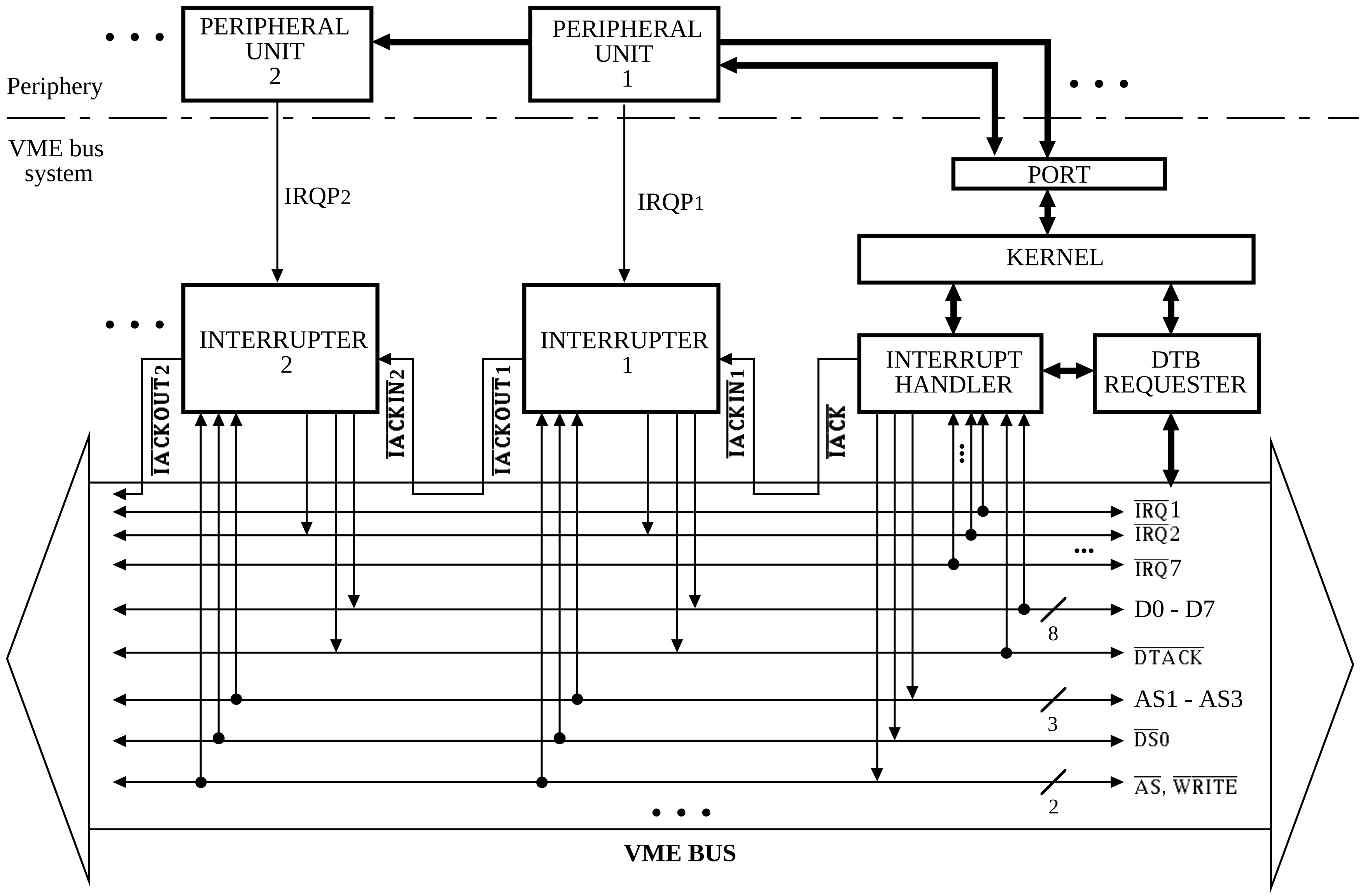


**Figure 1** The interrupt system

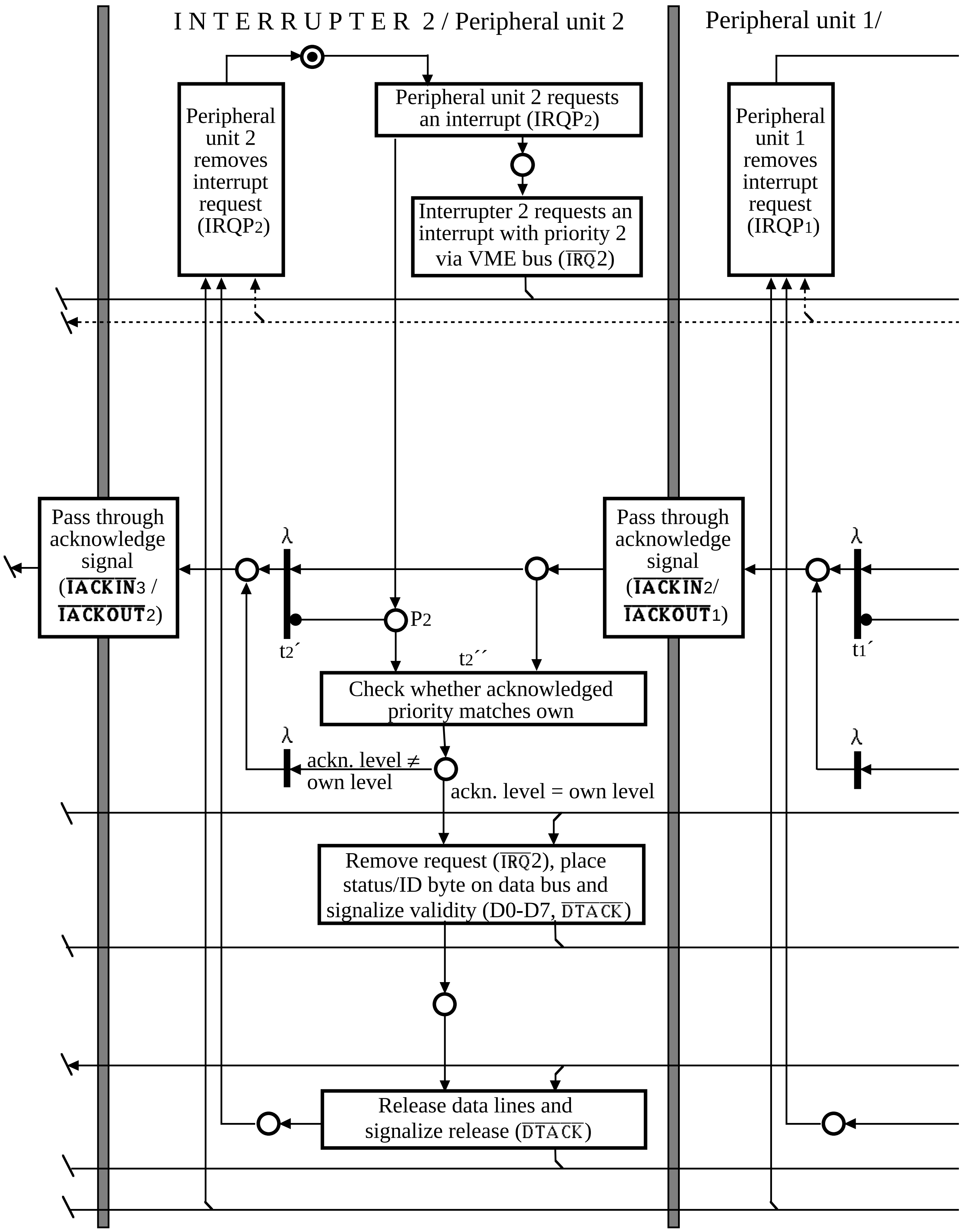

I N T E R R U P T E R 2 / Peripheral unit 2
Peripheral unit 1/
Peripheral unit 2 removes interrupt request (IRQP2)
Peripheral unit 2 requests an interrupt (IRQP2)
Interrupter 2 requests an interrupt with priority 2 via VME bus (IRQ2)
Peripheral unit 1 removes interrupt request (IRQP1)
Pass through acknowledge signal (IACKIN3 / IACKOUT2)
λ
Pass through acknowledge signal (IACKIN2/ IACKOUT1)
λ
P2
t2´
t2´´
t1´
Check whether acknowledged priority matches own
λ
ackn. level ≠ own level
ackn. level = own level
λ
Remove request (IRQ2), place status/ID byte on data bus and signalize validity (D0-D7, DTACK)
Release data lines and signalize release (DTACK)

**Figure 2 (left part)** High-level specification

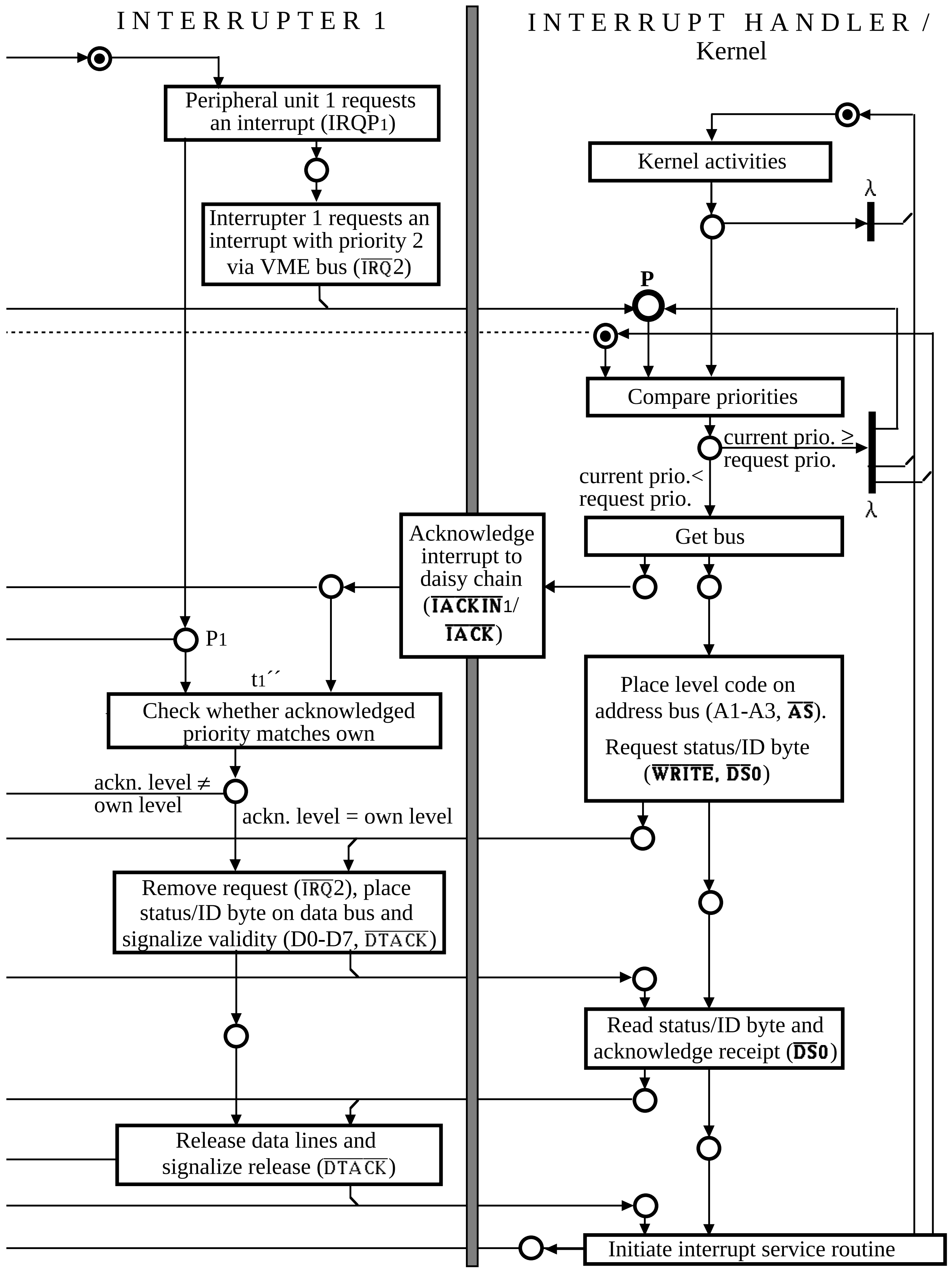

INTERRUPTER 1
INTERRUPT HANDLER / Kernel
Peripheral unit 1 requests an interrupt (IRQP1)
Interrupter 1 requests an interrupt with priority 2 via VME bus (IRQ2)
Kernel activities
λ
P
Compare priorities
current prio. ≥ request prio.
current prio.< request prio.
λ
Get bus
Acknowledge interrupt to daisy chain (IACKIN1/ IACK)
P1
t1´´
Check whether acknowledged priority matches own
ackn. level ≠ own level
ackn. level = own level
Place level code on address bus (A1-A3, AS).
Request status/ID byte (WRITE, DS0)
Remove request (IRQ2), place status/ID byte on data bus and signalize validity (D0-D7, DTACK)
Read status/ID byte and acknowledge receipt (DS0)
Release data lines and signalize release (DTACK)
Initiate interrupt service routine

**Figure 2 (right part)** High-level specification

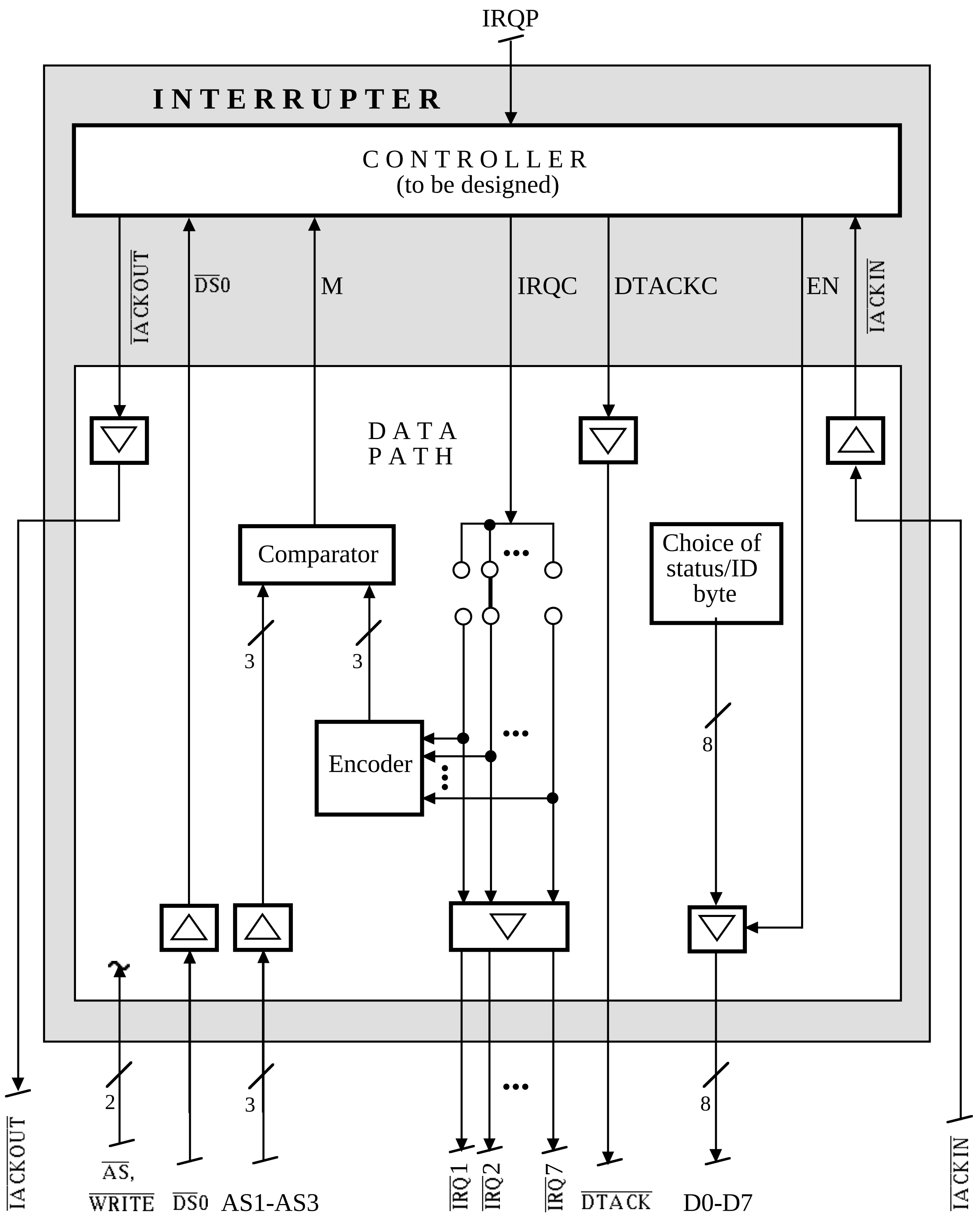
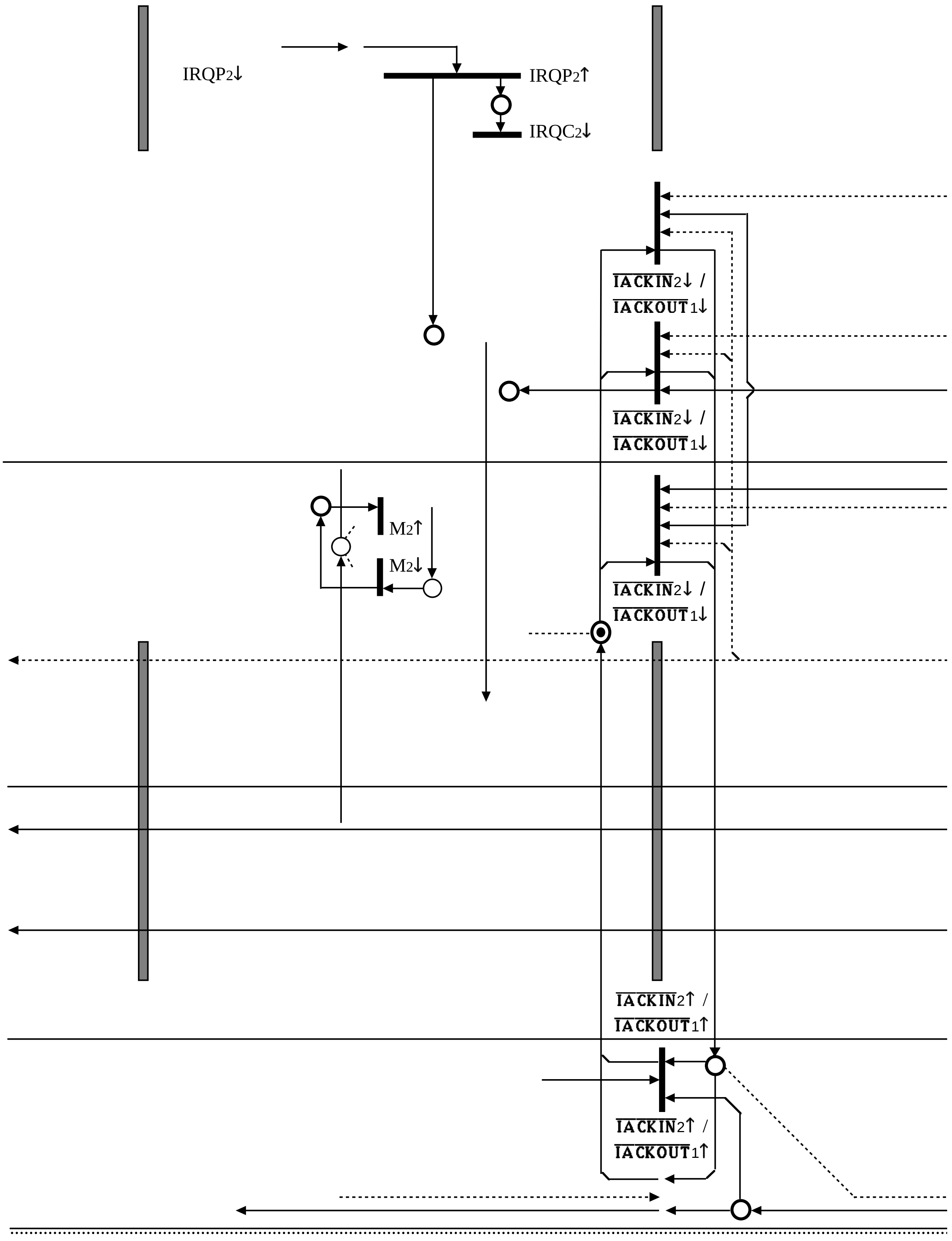
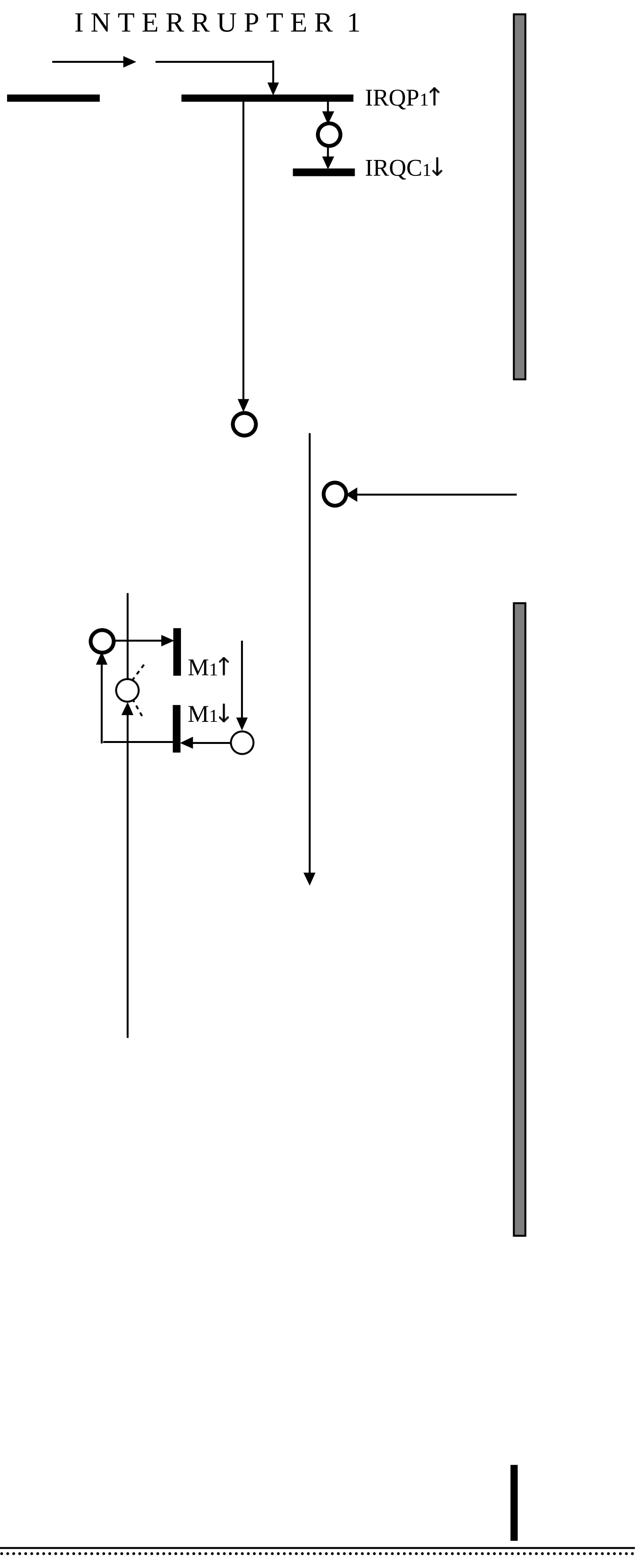
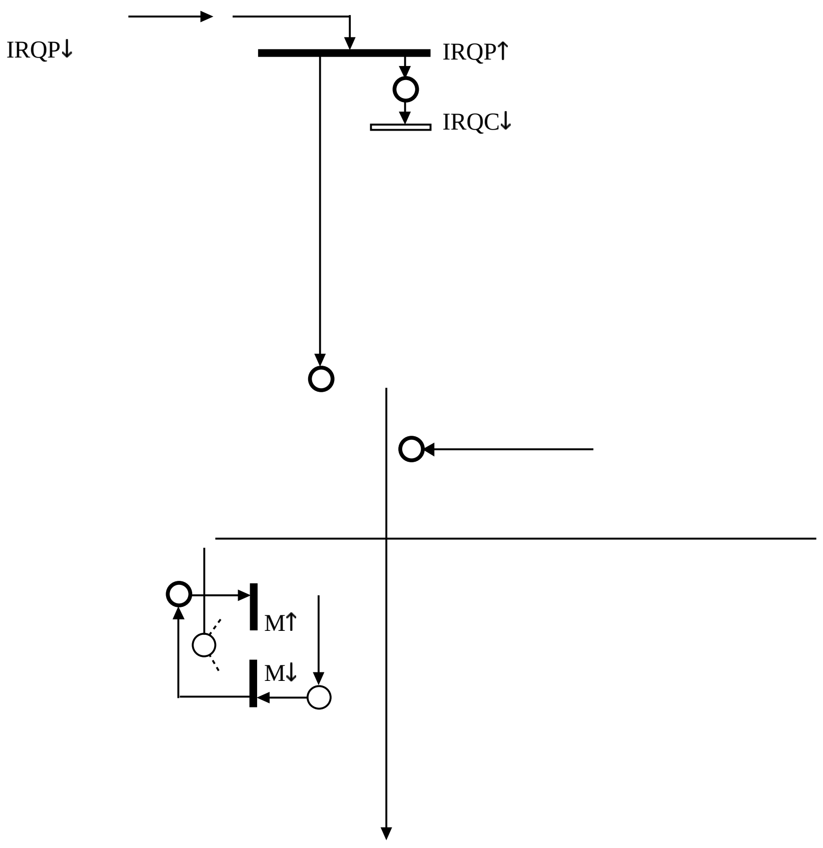


**Figure 3** An interrupter: data path and controller


IRQP2↓
IRQP2↑
IRQC2↓
IACKIN2↓ / IACKOUT1↓
IACKIN2↓ / IACKOUT1↓
M2↑
M2↓
IACKIN2↓ / IACKOUT1↓
IACKIN2↑ / IACKOUT1↑
IACKIN2↑ / IACKOUT1↑


INTERRUPTER 1
IRQP1↑
IRQC1↓
M1↑
M1↓


IRQP↓
IRQP↑
IRQC↓
M↑
M↓

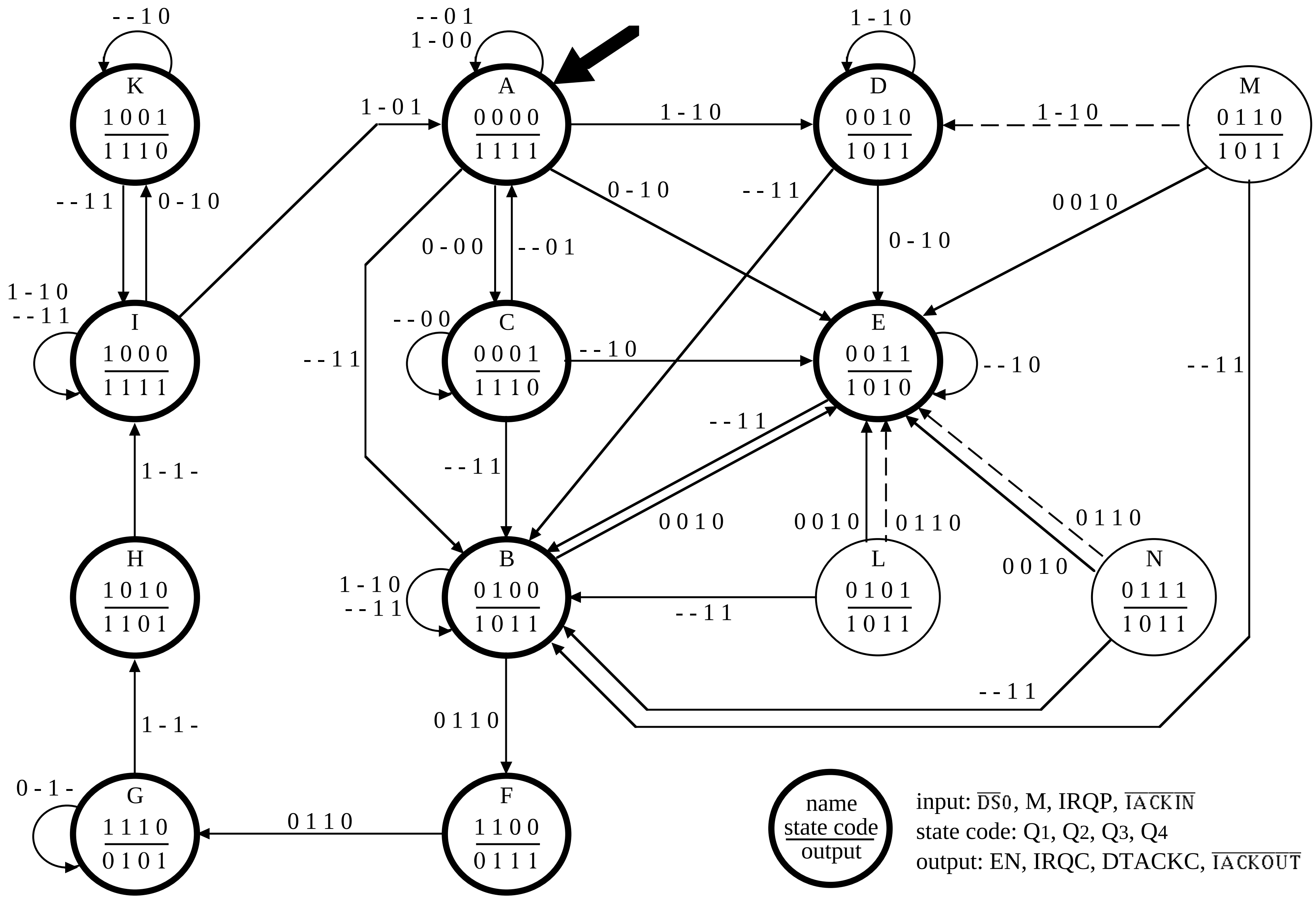


**Figure 6** Reduced state machine of the controller with non-fundamental mode state assignment