The Dissertation Committee for Jianping Song
certifies that this is the approved version of the following dissertation:

Constraint-based Real-time Scheduling for Process Control

Committee:

Aloysius K. Mok, Supervisor

James C. Browne

Mohamed G. Gouda

Yin Zhang

Deji Chen

Constraint-based Real-time Scheduling for Process Control

by

Jianping Song, M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2010

Acknowledgments

This dissertation is about task scheduling. Scheduling algorithms, no matter how powerful they are, cannot handle every workload correctly without a set of conditions guaranteed by the tasks and resources. The same is true with an academic endeavor. I would like to provide an incomplete account of the privileges and favors I got from other people here.

Thanks to my advisor, Professor Aloysius K. Mok, for his patience, support, and inspiration. He is my research advisor, my mentor and my friend. He gave me much needed advice and help when I felt frustrated, whether on research or in life. I will leave this research group with a great appreciation of being helped.

Thanks to my committee, Professor James Browne, Professor Mohamed Gouda, Professor Yin Zhang, and Dr. Deji Chen. Their comments make a positive and significant influence to this research. A special thanks goes to Dr. Deji Chen, an expert in industrial control and real-time scheduling, who provides numerous inputs during my research.

I cannot thank Mark Nixon enough for his invaluable initiative and support. He got involved in some of my projects and gave me numerous insightful suggestions. More important, it is him who makes some experiments in this dissertation possible. I am also deeply in debt to Terry Blevins, who

helped me understand the details of the PID algorithm.

Thanks for all group members who helped me along the way. Special thanks to two group members who work on the same project, Song Han and Xiuming Zhu. Oftentimes, we three sit together to fix some nasty bugs. It has been a pleasure to collaborate with these two talented guys.

Thanks to my father Niangui Song and my mother Shenglan Gao for their support and patience. They teach me to be a person with integrity, responsibility, and perseverance. Everything I have accomplished, I owe to them. Thanks also to my sisters, who believe in me and support whatever I do.

Constraint-based Real-time Scheduling for Process Control

Publication No. _____

Jianping Song, Ph.D.
The University of Texas at Austin, 2010

Supervisor: Aloysius K. Mok

This research addresses real-time task scheduling in industrial process control. It includes a constraint-based scheduler which is based on MSP.RTL, a tool for real-time multiprocessor scheduling problems with a wide variety of timing constraints. This dissertation extends previous work in two broad directions: improving the tool itself and broadening the application domain of the tool to include wired and wireless industrial process control. For the tool itself, we propose enhancements to MSP.RTL in three steps. In the first step, we modify the data structure for representing the temporal constraint graph and cutting the memory usage in half. In the second step, we model the search problem as a constraint satisfaction problem (CSP) and utilize backmarking and conflict-directed backjumping to speed up the search process. In the third step, we perform the search from the perspective of constraint satisfaction programming. As a result, we are able to use existing CSP techniques efficiently, such as look ahead, backjumping and consistency checking. Compared to the

various ad hoc heuristics used in the original version, the new approach is more systematic and powerful.

To exercise the new MSP.RTL tool, we acquired an updated version of the Boeing 777 Integrated Airplane Information Management System(AIMS). This new benchmark problem is more complicated than the old one used in the original tool in that data communications are described in messages and a message can have multiple senders and receivers. The new MSP.RTL tool successfully solved the new benchmark problem, whereas the old tool would not be able to do so.

In order to apply real-time scheduling in industrial process control, we carry out our research in two directions. First, we apply the improved tool to traditional wired process control. The tool has been successfully applied to solve the block assignment problem in Fieldbus networks, where each block comprising the control system is assigned to a specific device such that certain metrics of the system can be optimized. Wireless industrial control has received a lot of attention recently. We experimented with the tool to schedule communications on a simulated wireless industrial network.

In order to integrate the scheduler in real wireless process control systems, we are building an experimental platform based on the WirelessHART standard. WirelessHART, as the first open wireless standard for process control, defines a time synchronized MAC layer, which is ideal for real time process control. We have successfully implemented a prototype WirelessHART stack on Freescale JM128 toolkits and built some demo applications on top of it.

Even with the scheduler tool to regulate communications in a wireless process control, it may still be possible that communications cannot be established on an inferior wireless link within an expected period. In order to handle this type of failures, we propose to make the control modules aware of the unreliability of wireless links, that is, to make the control modules adapt to the varying link qualities. PID(Proportional, Integral, Derivative) modules are the most used control modules. We developed PIDPlus, an enhanced PID algorithm to cope with possible lost inputs and outputs. It has been shown that PIDPlus can drastically improve the stability of the control loop in cases of unreliable wireless communications.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Traditionally, schedulability analysis focuses on a scheduling model (e.g., the Lib & Lapland periodic task model [36]) and performs analysis (e.g., RAM) on input task sets which conform to the model [28, 37, 38, 40, 45]. This paradigm has proven effective in practice as long as the application can be abstracted to fit the scheduling model. In case of a misfit, the schedulability analysis tool cannot be used. In contrast, the MSP.RTL tool, first reported in [39], follows a drastically different paradigm: it treats real-time scheduling as a model construction problem from the formal specification of timing constraints, specifically RTL (Real Time Logic) [31] formulas.

The input to MSP.RTL can be a wide variety of timing requirements, as long as the semantics of those requirements can be translated into RTL formulas. The same basic synthesis engine will be used for all application domains, with the long-term goal of incorporating more powerful results from real-time scheduling theory to be used as domain-specific heuristics for the synthesis engine.

MSP.RTL computes cyclic schedules for multiprocessor systems in a three-step process: the first step (Initialization) constructs a temporal con-

straint graph representing the input timing specification. The second step (Phase 1) finds a set of solutions of the temporal constraint graph by using a combination of logic reasoning and positive cycles detection. The third step (Phase 2) searches the set of solutions to find a feasible schedule that satisfies the resource constraints by various search strategies and by exploiting techniques from real-time scheduling theory.

This dissertation extends previous work in two broad directions: improving the tool itself and broadening the application domain of the tool to include wired and wireless industrial process control. For the tool itself, we propose enhancements to MSP.RTL in three steps. During initialization, we modify the data structure for representing the temporal constraint graph and cutting the memory usage in half. In Phase 1, we model the search problem as a constraint satisfaction problem (CSP) that utilizes backmarking and conflict-directed backjumping to speed up the search process. In Phase 2, we perform the search from the perspective of constraint satisfaction programming. As a result, we are able to use standard CSP search techniques efficiently, such as backjumping and consistency checking. In the process of applying CSP search techniques, we can apply some domain specific results from real-time scheduling theory. Compared to the various ad hoc heuristics used in the original version, the new approach is neater and more systematic.

This dissertation first presents the improvements to the MSP.RTL tool. Then we describe its application to a sanitized version of the Boeing 777 Integrated Airplane Information Management System (AIMS). The AIMS sys-

tem [14], running on the ARINC 659 [29], requires high resource utilization and performance guarantees while providing strict partitioning of functions on a multiprocessor platform. The scheduling problem involves pre-scheduling of both computational and communication resources and involves both deadline and relative timing requirements. For example, a relative timing constraint may be imposed on successive instances of a periodic task such that the two instances must be separated by a minimum interval. Communication latency constraints are also needed to make sure that the interval between the completion of the receiver task and the start of the sender task does not exceed a specified value. Compared to the data set in [39], this data set describes communications in data messages, instead of sending/receiving process pairs. Specifically, for a datum, the input specification describes the length and whether it appears on the bus. Then, a process lists all data items it produces and consumes (which implies these processes are not independent). It may happen that one datum has several senders and receivers. These changes pose a big challenge for our search algorithm.

The other broad direction for extending previous work is to apply the improved tool to industrial process control. In industrial process control, new types of constraints require augmentation to the MSP.RTL tool. Foundation Fieldbus (FF) is one of the most popular wired process control network standards. In a Foundation Fieldbus-based process control system, control strategies are implemented in a distributed environment on devices connected to the fieldbus[50]. A control strategy is a policy to instruct a process to reach

desired states, such as maintaining a tank level at a specified range. To implement a strategy, Foundation Fieldbus defines a plethora of building blocks which are known as function blocks in the specification, such as input blocks, output blocks and control blocks (implementations of control algorithms). To deploy a control strategy on a fieldbus, a system engineer first assigns function blocks to devices, then designates the start time of each block. This deployment procedure is time-consuming and may not achieve the best result possible. With the help of MSP.RTL, we want to configure a control strategy automatically among the fieldbus devices optimally as measured by the overall finish time and the number of published messages on the bus [51].

Wireless process control has been a popular topic recently in the field of industrial control. Compared to traditional wired process control systems, their wireless counterparts have the potential to save cost and simplify installation. As a result, several industrial organizations, such as ISA [30], HART [27], and ZigBee [56], have been actively pushing the application of wireless technologies in industrial automation. ZigBee Specification V1.0 was ratified in late 2004 and ZigBee compliant products are readily available on the market. WirelessHART is released by the HART Communication Foundation in September 2007. Notably, WirelessHART is the first open wireless communication standard specifically designed for process measurement and control applications.

Nevertheless, wireless process control faces some big challenges: 1) Unreliable transmissions, 2) Limited power supply, and 3) Security. Of the

three challenges, unreliable transmissions and limited power supply can affect scheduling decisions profoundly. Transmission failures can be due to various factors, such as interferences and obstacles. Problematic transmissions could cause the pre-computed schedule to fail to execute. In a wireless process control system, some nodes may be battery-powered. It would be ideal if the scheduler can take into account the energy consumption. Meanwhile, the resulting schedule should be robust enough to react properly in case a node runs out of power. In this research, we are able to use our tool to schedule communications on a simulated ZigBee-based wireless industrial network [50] to meet some of the challenges listed above.

In order to integrate the scheduler in real wireless process control systems, we are building an experimental platform based on the WirelessHART standard. WirelessHART [27], as the first open wireless standard for process control, defines a time synchronized MAC layer, which is ideal for real time process control. We have successfully implemented a prototype WirelessHART stack on Freescale MCF51JM128 toolkits and built some demo applications on top of it [47].

Even with our scheduler, it may still be possible that communications cannot be reestablished on a wireless link within an expected period. In order to handle this type of failures, we propose to make the control modules aware of the unreliability of a wireless link, that is, to improve the control modules. PID (Proportional, Integral, Derivative) modules [33] are the most used control modules. We developed PIDPlus [52], an enhanced PID algorithm to

cope with possible lost inputs and outputs. It has been shown that PIDPlus can drastically improve the stability of the control loop in cases of unreliable wireless communications.

1.1 Related Work

Generally, the problem of real-time multiprocessor scheduling is *NP-hard* [25]. Current schemes for real-time multiprocessor scheduling can be categorized into two approaches: partitioning and global scheduling [13]. In partitioning schemes, tasks are statically assigned to processors and all jobs (a job is an instance of a task) of a task are executed on the same processor. In contrast, in global scheduling schemes, all eligible jobs are stored in a single priority queue; the global scheduler selects for execution the highest priority jobs from the queue. Thus a task can migrate from one processor to another during the execution of different jobs. The MSP.RTL tool uses a partitioning scheme.

In most existing partition schemes [16, 41, 42], communications between tasks are not considered, i.e., these tasks are independent. The work that is most similar to ours is KRONOS[9]. The authors take a different approach. Given a timed automata with deadlines (TAD) A and a property P, they try to derive a TAD T_P that describes all the schedules that satisfy the property P. Due to state space explosion problem, their approach is only suitable for trivial systems of small size.

Our work benefits much from temporal constraint graphs[18] and con-

straint satisfaction programming [17]. Meanwhile, MSP.RTL also uses results from real-time scheduling theory. In this way, MSP.RTL can solve a wide variety of scheduling problems efficiently.

The rest of this dissertation is organized as follows. Chapter 2 gives a description of the input to MSP.RTL and a definition of the problem. Chapter 3 describes the scheduler synthesis engine of the new MSP.RTL tool, focusing on the improvements we implemented. Chapter 4 discusses the applications of the MSP.RTL tool to wired and wireless industrial networks. The experimental WirelessHART platform is discussed in Chapter 5. Chapter 6 details the PIDPlus control module and its advantage over standard PID modules. Chapter 7 gives some directions for future work and concludes this dissertation.

Chapter 2

Problem Definition

The input to the MSP.RTL tool is a set of Real-Time Logic (RTL [31]) formulas. For each application domain, however, a translator may be provided to convert the particular syntax of the application requirements to RTL. MSP.RTL would then instantiate these formulas to create a scheduling problem definition. By having RTL as a base language, all timing constraints can be treated in a uniform manner. We give a brief description of RTL In the following section.

2.1 Real-Time Logic (RTL)

RTL [31] is a multi-sorted first-order logic. A computation in RTL is a sequence of event sets. Time passes between sets of events, and an event occurrence marks a point in time. An action is an activity which requires a non-zero but bounded amount of system resources. The execution of an action is represented by two events: one denoting its initiation and the other denoting its termination [39].

- *Start and Stop Events*: We use the notation $\uparrow A$ to represent the event marking the initiation of action A, and $\downarrow A$ to denote the event marking

the completion of action A.

- *Occurrence function*: The occurrence function, denoted by the character @, is introduced to capture the notion of real time.

$$@(e, i) \equiv \text{time of the } i\text{th occurrence of event } e.$$

For example, for a periodic task A whose period is $t[A]$ and computation time is $c[A]$ we have the following RTL formula:

$$\begin{array}{lll} \forall i \geq 1 & & @(\uparrow A, i) \geq (i-1) \times t[A] \\ & \wedge & @(\downarrow A, i) \leq i \times t[A] \\ & \wedge & @(\uparrow A, i) + c[A] \leq @(\downarrow A, i) \end{array}$$

2.2 Input to the Synthesis Engine

The RTL input is analyzed by the MSP.RTL tool to output a run-time scheduler. MSP.RTL outputs cyclic executives for a multiprocessor implementation environment by sequencing events within a major frame, where the length of a major frame is the least common multiple (LCM) of all periods of all tasks. Within a major frame, each task is instantiated to one or more instances, where each instance is called a *job*. For example, if the major frame is $2 \times t[A]$, then task A is instantiated into two jobs *A:1* and *A:2*. Each job is executed only once. For brevity, we use $\uparrow$*A:1* to denote the time of the occurrence of the start of task instance (job) *A:1*. The above formula reduces

to the following instances:

$$\begin{array}{ll} & \uparrow A:1 \geq 0 \\ \wedge & \downarrow A:1 \leq t[A] \\ \wedge & \uparrow A:1+c[A] \leq \downarrow A:1 \\ \wedge & \uparrow A:2 \geq t[A] \\ \wedge & \downarrow A:2 \leq 2 \times t[A] \\ \wedge & \uparrow A:2+c[A] \leq \downarrow A:2 \end{array}$$

The result of instantiating the input RTL formulas is a set of ground constraints which are linear inequalities involving event instances. An instance of a task consists of two event instances, its start and stop events. MSP.RTL has a textual representation of these ground constraints. Program 1 is a simple problem illustrating the syntax.

2.3 The Satisfiability Problem

In general, the timing constraints of the system take the form

$$D_1 \wedge D_2 \wedge \cdots \wedge D_m \tag{2.1}$$

where each D_i is a clause of the form

$$C_1 \vee C_2 \vee \cdots \vee C_n \tag{2.2}$$

, each C_j is a clause of the form

$$L_1 \wedge L_2 \wedge \cdots \wedge L_l \tag{2.3}$$

, and each L_k is a literal of one of the following forms

$$v_1 \pm I \leq v_2 \tag{2.4}$$

Program 1 Example of a scheduling problem definition used by MSP.RTL.

```
PROBLEM msp1
PROCESSOR P1

JOBS
/*  assigned   computation  release  deadline */
/*  processor  time         time     time     */
A     (P1)     20           40       110
B     (P1)     20           60       90
C     (P1)     20           50       91
D     (P1)     20           0        120

CONSTRAINTS
    { /* A and B are mutually exclusive */
        [_A <= ^B]
       OR
        [_B <= ^A]
    }
AND { /* B and C are mutually exclusive */
        [_B <= ^C]
       OR
        [_C <= ^B]
    }
AND { /* A and C are mutually exclusive */
        [_A <= ^C]
       OR
        [_C <= ^A]
    }
```

$$v_1 \leq I \tag{2.5}$$

$$I \leq v_1 \tag{2.6}$$

where v_1 and v_2 can be either the start time or the finish time of a job, and I is an integer constant. Basically, we only allow two event occurrences in an inequality. In addition, we assume system time is measured in integer multiples of time units. That is, the start and end time of a job are always integers. Our goal is to find a feasible schedule whenever one exists such that each job starts executing after its release time and completes its computation before its deadline, consistent with the specified timing and resource constraints.

Chapter 3

The New MSP.RTL Algorithm

To compute a multiprocessor schedule, MSP.RTL needs to solve a satisfiability problem for the ground instances of the formulas resulting from instantiating the input RTL formulas. The MSP.RTL algorithm consists of three parts: the first part (Initialization) constructs a temporal constraint graph representing the basic input timing specification. The second part (Phase 1: solving timing constraints) finds a set of solutions of the temporal constraint graph by using constraint satisfaction programming techniques. The third part (Phase 2: solving resource constraints) searches the set of solutions to find a feasible schedule which satisfies the resource constraints by using a combination of constraint programming strategies and results from real-time scheduling theory.

The skeleton of this algorithm is shown in Algorithm 1.

3.1 Temporal constraint graph

The timing constraint of the system is in the form of Equation (2.1). The clauses in Equation (2.1) can be partitioned into two sets, a set of *unit clauses* and a set of *disjunctive clauses*. Each clause in the set of unit clauses

Algorithm 1 The New MSP.RTL algorithm

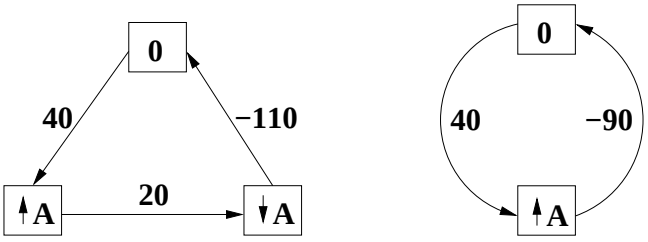
```
function SCHEDULE(J, C)      ▷ J: set of jobs; C: set of timing constraints
    construct the temporal constraint graph G from J and C
    while true do
        find next satisfiable solution to G
        if found then
            find a feasible schedule by enforcing the resource constraints
            if found a schedule then
                return the final schedule
            end if
        end if
        return error
    end while
end function
```

contains conjunctive clauses whose underlying literals are in the form given by one of Equations (2.4)–(2.6). By mathematical logic, unit clauses must be satisfied whereas one of the clauses in a disjunctive clause needs to be satisfied. We associate the set of unit clauses with a directed graph with weighted edges called a *temporal constraint graph*. In this graph, vertices denote the terms which are not integer constants; each edge, $v_i \rightarrow v_j$, is labeled by a weight a_{ij}, representing the literal $v_i + a_{ij} \leq v_j$. In order to provide a system time reference point, a special node with label **0** is introduced to represent the "beginning of the world" - time 0. As we focus on non-preemptive scheduling in this research, we always have

$$\downarrow A = \uparrow A + c[A] \tag{3.1}$$

Based on Equation (3.1), we can substitute the completion time of an action with its start time. Correspondingly, in the temporal graph, the vertex as-

sociated with the completion time of an action can be removed. The edges incident to that vertex are redirected to the vertex associated with the start time of that action, with the weights adjusted accordingly. As a result, this substitution can cut the number of vertices in the constraint graph by half. For example, we can represent the release and deadline constraints on job A in Program 1 by a temporal constraint graph as shown in Figure 3.1.



(a) Before Substitution

(b) After Substitution

Figure 3.1: Release and deadline constraints on job A

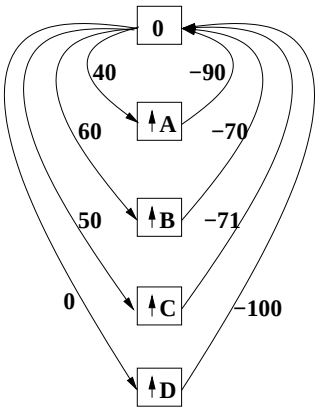
As the temporal constraint graph captures only the constraints represented by the set of unit clauses, we also keep a *disjunctive clause list* to store the set of disjunctive clauses.

As shown in Figure 3.2, Program 1 can be represented by a constraint graph and a list of disjunctive clauses.

3.1.1 Positive cycles and unsatisfiability

The goal of this step is to check the satisfiability of temporal constraints of the given problem.

In a temporal constraint graph G, each edge, $v_i \rightarrow v_j$, is labeled by a



Disjunctive clauses:

$\uparrow A + 20 \leq \uparrow B$	$\vee$	$\uparrow B + 20 \leq \uparrow A$
$\uparrow B + 20 \leq \uparrow C$	$\vee$	$\uparrow C + 20 \leq \uparrow B$
$\uparrow A + 20 \leq \uparrow C$	$\vee$	$\uparrow C + 20 \leq \uparrow A$

Figure 3.2: Constraint graph for Program 1

weight a_{ij}, representing the linear inequality $v_i + a_{ij} \leq v_j$. each path from i to j in G, $i_0 = i, i_1, \ldots, i_k = j$, induces the following constraint:

$$v_i + \sum_{j=1}^{k} a_{i_{j-1}, i_j} \leq v_j$$

If there is more than one path from v_i to v_j, then it can be easily verified that the conjunction of all the induced path constraints yields

$$v_i + d_{ij} \leq v_j$$

where d_{ij} is the length of the *longest* path from v_i to v_j.

Consequently, d_{0j} is the lower bound for v_j and $-d_{i0}$ is the upper bound for v_i.

In [32], Jahanian and Mok proposed a procedure for deciding the unsatisfiability of a restricted subset of RTL formulas by means of positive cycle detection in the temporal constraint graph. If a cycle with *positive* weight exists in the constraint graph G, the formula P consisting of the conjunction of literals (inequalities) which correspond to the edges involved in the cycle is unsatisfiable. In the ground case, the temporal constraint of a given problem is consistent if and only if its constraint graph, G, has no positive cycle.

If the temporal constraints of the given problem are inconsistent, we can stop and conclude that there does not exist any feasible schedule for the problem. If the constraint graph is *consistent*, the following theorem holds.

Theorem 1 ([39]). *If a constraint graph G is consistent, then the set of feasible values for variable v_i is in the range $[d_{0i}, -d_{i0}]$, where d_{0i} is the length of the longest path from special node* **0** *to v_i and d_{i0} is the length of the longest path from v_i to special node* **0**.

$d_{01}, \ldots, d_{0n}$ and $d_{10}, \ldots, d_{n0}$ can be calculated by applying the single source longest path algorithm of Bellman-Ford [22] to the constraint graph to find the longest distances to/from all vertices from/to node **0**, respectively. The bound on the number of operations is $O(|E| \times |V|)$, where $|E|$ is the number of edges in the constraint graph and $|V|$ is the number of vertices in the constraint graph.

3.2 Phase 1: Search for a satisfiable solution to the temporal constraint graph

In general, there may be disjunctive clauses in the constraint formulas. We only need to satisfy one of the clauses in a disjunctive clause to satisfy the disjunctive clause. We define a *labeling* of the disjunctive clauses as a selection of one clause from each disjunctive clause. Each labeling defines a constraint graph which includes constraints in the selected clauses and unit clauses. The original problem is consistent if and only if there is a labeling whose associated constraint graph is consistent.

The satisfiability problem can be mapped to a constraint satisfaction problem (CSP) as follows. Each disjunctive clause in the constraint formulas is associated with a variable x in the resulting CSP problem, whose domain is the set of conjunctive clauses in that disjunctive clause. By $x = c_i$ we denote that variable x is instantiated and the constraints in clause c_i is added to the constraint graph. A partial assignment $((x_1, c_1), \cdots, (x_i, c_i))$ corresponding to a partial labeling is consistent if and only if there is no positive cycle in the resulting temporal constraint graph. For a given variable (disjunctive clauses), its values (conjunctive literals) are tried in a fixed order.

The search process with backtracking in constraint satisfaction programming is as follows:

1. If all variables have been instantiated, return

2. If current variable x_i's domain is not empty, assign to it the next available

value

3. (**Backtracking**) Otherwise, set $i = i - 1$, go to step (2)

4. If current partial assignment is consistent, set $i = i + 1$, go to step (1)

5. Otherwise, go to step (2)

The basic technique to solve a CSP problem is backtracking. The backtracking algorithm instantiates the variables in a fixed order (variable ordering). Suppose the variable ordering is $x_1, \ldots, x_n$, we use a vector $\vec{c_i} = (c_1, \ldots, c_i)$ to denote the partial assignment $((x_1, c_1), \cdots, (x_i, c_i))$. If all values of variable x_i cannot lead to a solution, backtracking will go back to try the next value of variable x_{i-1}. In the worst case, we need to enumerate all labelings to find all satisfiable ones. To speed up the search process, we use two powerful constraint satisfaction techniques [17]:

- Backmarking: prune the search tree by effectively using the results of positive cycle detection
- Backjumping: avoid discovering repeated failures due to the same reason

3.2.1 Backmarking

Backmarking [17] prunes the search tree by effectively using the results of positive cycle detection. Suppose variable x_i is the current variable to be instantiated and is assigned a value of c_i. MSP.RTL checks if there is any positive cycle in the resulting constraint graph. If no positive cycle is found,

the current partial assignment is consistent. Otherwise, the partial assignment is inconsistent. In this case, MSP.RTL analyzes the positive cycle to derive a set of conflicting assignments. The conflicting assignment set is inserted into a *NOGOOD* assertion list as an item.

We use Program 1 to illustrate this technique. While choosing $\uparrow A{+}20 \leq \uparrow B$ and $\uparrow B{+}20{\leq}\uparrow C$ as a partial assignment, we add two edges $\uparrow A \xrightarrow{20} \uparrow B$ and $\uparrow B \xrightarrow{20} \uparrow C$ to the constraint graph and find a positive cycle in the constraint graph:

$$(0 + 60 \leq \uparrow B) \wedge (\uparrow B + 20 \leq \uparrow C) \wedge (\uparrow C - 71 \leq 0) \tag{3.2}$$

Since only $\uparrow B + 20 \leq \uparrow C$ belongs to a disjunctive clause, Equation (3.2) can be reduced to:

$$\uparrow B + 20 \leq \uparrow C \tag{3.3}$$

The effect is that $\uparrow B{+}20 \leq \uparrow C$ will be added to the *NOGOOD* list. The search procedure will prune any subtree whose root node is labeled with $\uparrow B{+}20 \leq \uparrow C$.

3.2.2 Conflict-based Backjumping

In the backtracking algorithm, a dead-end occurs if variable x_i has no consistent values left, in which case backtracking will go back to variable x_{i-1}. If there is no constraint between x_{i-1} and x_i, the same dead-end will be reached at x_i for each value of x_{i-1}. Backjumping [17] schemes are one of the primary tools for reducing the rediscovering of the same dead-ends in backtracking.

Whenever backjumping finds a dead-end, it tries to jump as far back as possible without skipping potential solutions. Two terms are used to describe

a jump precisely: safety in jumping and optimality in the magnitude of the jump.

Definition 1 (**i-leaf dead-ends**[17]). *Given a variable ordering* $d = x_1, \cdots, x_n$, *then a partial assignment* $\vec{c_i} = (c_1, \ldots, c_i)$ *that is consistent but is in conflict with* x_{i+1} *is called an* $i-$*leaf dead-end state.* x_{i+1} *is called a dead-end variable.*

Definition 2 (**safe jump**[17]). *Let* $\vec{c_i} = (c_1, \ldots, c_i)$ *be an* $i-$*leaf dead-end state. We say that* x_j, *where* $j \leq i$, *is safe if the partial assignment* $\vec{c_j} = (c_1, \ldots, c_j)$ *cannot be extended to a solution.*

Definition 3 (**optimal jump**). *Let* $\vec{c_i} = (c_1, \ldots, c_i)$ *be an* $i-$*leaf dead-end state. We say that* x_j, *where* $j \leq i$ *is optimal if* x_j *is safe and for any* $k < j$, x_k *is not safe.*

Ideally, whenever a dead-end is reached, we should make an optimal jump, which can reduce all unnecessary retries without missing any solution.

We implement a specific backjumping scheme called *conflict-directed backjumping* [17], where each variable is associated with a jumpback set. Whenever a dead-end is reached, the algorithm selects from the jumpback set (associated with the dead-end variable) the variable with highest rank based on variable ordering and jumps to it. Hence central to this algorithm is the construction of the jumpback sets, as presented in Algorithm 2, where G is the base temporal constraint G constructed with the unit clauses and D is a list of disjunctive clauses $D_1, \cdots, D_n$.

Algorithm 2 Conflict-directed backjumping in MSP.RTL

```
1: function LabelingSearch(G, D)
2:     cur ← 0
3:     while cur ≠ n do
4:         cur ← cur +1
5:         nc ← number of conjunctive clauses in D_cur
6:         ci = 0
7:         while ci ≠ nc do
8:             ci ← ci + 1
9:             Add constraints in C_{cur,ci} to G, resulting G'
10:            if there is a positive cycle P in G' then
11:                Add the indices of disjunctive clauses participating in P to
   J_cur
12:                Restore G' to G
13:                continue
14:            end if
15:            break
16:        end while
17:        if ci == nc then
18:            if J_cur = ∅ then
19:                break;
20:            end if
21:            J_temp ← J_cur
22:            cur ← the largest number in J_temp
23:            J_cur ← J_cur ∪ J_temp − {cur}
24:        end if
25:    end while
26:    if cur == n then
27:        return true
28:    end if
29:    return false
30: end function
```

The core data structure in the algorithm is the jumpback set J_{cur}. In a standard conflict-directed backjumping scheme, Line (22) would retrieve a safe and optimal jump target. However, due to the complexity in detecting positive cycles, the jump back target in our implementation is safe but not optimal, which means this algorithm can reach the same dead-ends several times as well. When a conjunctive clause is added to the temporal graph G, it may create several positive cycles.

We use an example to illustrate how conflict-directed backjumping works in MSP.RTL. Suppose the 68-th disjunctive clause has only two clauses, $C_{68,1}$ and $C_{68,2}$. When trying $C_{68,1}$, MSP.RTL detects a positive cycle

$$C_{3,1} \rightarrow C_{7,2} \rightarrow C_{18,1} \rightarrow C_{68,1}$$

. Next $C_{68,2}$ is tried. Another positive cycle

$$C_{6,1} \rightarrow C_{9,1} \rightarrow C_{68,2}$$

is found. A dead-end occurs at variable 68. The jumpback set of variable 68 is $\{3, 6, 7, 9, 18\}$. Then the backjumping algorithm jumps back to variable 18. By changing the clause selected in the disjunctive clause 18, we may be able to solve the dead-end at the disjunctive clause 68.

Note that the backjumping in our implementation is not optimal as the algorithm used to detect positive cycles can only report one cycle at a time. We use the example above to explain this problem. Suppose there is another positive cycle

$$C_{10,1} \rightarrow C_{13,1} \rightarrow C_{68,1}$$

when $C_{68,1}$ is tried. This cycle is not reported by the positive cycle detection algorithm. In this case, we can jump further back to variable 13 without losing any solution.

3.3 Phase 2: Search for a feasible schedule

After a labeling is derived in Phase 1, we can calculate the valid domain for each vertex v_i in the temporal constraint graph by Theorem 1. If there are unlimited resources available, MSP.RTL can simply choose a value from the domain of each vertex to create a feasible schedule. However, as resources are limited, we also need to consider *resource constraints*. For example, no two jobs can be executed simultaneously on the same processor. The temporal constraint graph alone does not disallow this case since resource constraints are in addition to the logical temporal ordering requirements.

Note that $LabelingSearch()$, which solves the timing constraints, runs in polynomial time. It is the resource constraints that render this multiprocessor scheduling problem *NP-Hard* [25]. This is because a temporal constraint usually includes at most two jobs, whereas a resource constraint can include more than two jobs.

We use a combination of *scheduling-event-driven* search and constraint satisfaction programming to handle resource constraints. Here a scheduling event is the arrival of a job or the completion of a job. When a scheduling event occurs, a scheduling decision is made to pick a number of jobs to be executed subject to resource constraints. We call the time at which a scheduling decision

is made in response to events a *scheduling point*.

The search process involves scheduling of the jobs and updating of the domains of the associated vertices in the temporal constraint graph (denoted by $[lb, ub]$).

Each processor has three job lists: ineligible list, waiting list and ready list. Initially, all jobs on a processor are inserted into the ineligible list of that processor. A job moves from the ineligible list to the waiting list when all its predecessors have completed execution. A job in the waiting list moves to the ready list if its release time constraint is met. During initialization, the domain of node i, $(lb[i], ub[i])$, is initialized to $[d_{0,i}, -d_{i,0}]$.

Given a list of ready jobs, the order in which we consider them for scheduling will influence whether or not a feasible schedule can be found. In this research, we use SLST/MISF (Smallest Latest Start Time/Maximum Immediate Successors First) [45] heuristic. That is, the ready list is ordered in increasing latest start time of the jobs in the list. If there is a tie, the job with more successors is placed first. The outline of the scheduling algorithm is given in Algorithm 3.

Assume the associated vertex of job J_i is v. When job J_i is put on a processor for execution at schedule point t, vertex v is assigned a value of t. To maintain the semantics of the temporal constraint graph, the domain of each uninstantiated vertex reachable directly from/to v should be updated. This step is called *forward checking* in CSP and is implemented in function

Update_LBUB_Graph().

$$lb[j] = \begin{cases} \max(lb[j], t + a_{i,j}) & \text{if } v_j \in Succ(v_i) \\ lb[j] & \text{otherwise} \end{cases} \tag{3.4}$$

$$ub[j] = \begin{cases} \min(ub[j], t - a_{j,i}) & \text{if } v_j \in Pred(v_i) \\ ub[j] & \text{otherwise} \end{cases} \tag{3.5}$$

where:

$$Pred(x) = \{y | y \rightarrow x \in G\} \tag{3.6}$$

$$Succ(x) = \{z | x \rightarrow z \in G\} \tag{3.7}$$

From equations given above, we can see that function Update_LBUB_Graph() handles the temporal constraints. On the other hand, Update_LB_Successors() handles resource constraints. After J_i is started, the earliest time other jobs on the same processor can start is the completion time of J_i. This constraint is implicitly imposed by limited resources and is enforced by function Update_LB_Successors().

At a schedule point t, MSP.RTL performs the following consistency check:

- Suppose v is a vertex associated with an uncompleted job. Then $lb[v] <= ub[v]$. That is, vertex v must have a valid domain. If $lb[v] = ub[v]$, the corresponding job must start at $lb[v]$.
- Suppose v is a vertex associated with an uncompleted job. Then $ub[v] > t$. This check ensures a job would not miss its latest start time. If $t = ub[v]$, the corresponding job must have been started.

- The total time available on a processor in a time period is more than that required by the jobs on that processor which will execute in that period.

Whenever an inconsistency is found, we need to decide the schedule point and processor to jump back to. We cannot apply directly constraint satisfaction techniques such as conflict-directed backjumping in this step. The reason is that we have no way to identify the job that causes the inconsistency. For example, if the domain of job J_i is infeasible, we can make some changes to either decrease the lower bound or increase the upper bound. However, we can still create a jumpback set. MSP.RTL keeps track of the most recently scheduled jobs that determine job J_i's lower bound and upper bound. Then, for an infeasible domain of J_i, we have two jumpback candidates. Based on the SLST/MISF heuristic, MSP.RTL chooses the one with larger latest start time. Note that when a job is rescheduled, MSP.RTL would put another eligible job to run, instead of trying that job one time unit later. Thus this jumpback scheme is not safe, which means it may miss some solutions.

3.4 Benchmark Problem: The Boeing 777 AIMS Scheduling Problem

As a benchmark, we applied the new MSP.RTL to a sanitized version of the Boeing 777 integrated Airplane Information Management System(AIMS) [14]. The AIMS system consists of two redundant cabinets. In each cabinet, four processing modules, two spare processing modules and four

Algorithm 3 The Scheduling Algorithm with Resource Constraints

```
1: function ResourceSchedule(t_last, J_last) ▷ t_last: latest scheduling point
2:                                  ▷ J_last: the set of jobs to be executed at t_last
3:     if finished then
4:         return true
5:     end if
6:     find next schedule point t
7:     deduct (t − t_last) from the computation time of jobs in J_last
8:     for each job J in J_last do
9:         if J finishes at time t then
10:            ↓ J ← t
11:            delete J from the ready list
12:        end if
13:    end for
14:    move all waiting jobs with release time no later than t to the ready list
15:    while true do
16:        select the set of J_next from the ready lists
17:        for each job J in J_next do
18:            if J starts at time t then
19:                ↑ J ← t
20:                Update_LBUB_Graph(↑ J)
21:                Update_LB_Successors(↑ J)
22:            end if
23:        end for
24:        if CheckUpdatedBounds() == true then
25:            if ResourceSchedule(t, J_next) then
26:                return true
27:            end if
28:        end if
29:        if bt < t then
30:            remove the changes made by the scheduler during t_last, t
31:            break
32:        end if
33:        mark the ready list for processor 'pr' to ensure the next job in the
   list will be selected
34:    end while
35:    return false
36: end function
```

standard I/O modules share a common SAFEbus backplane [14, 29]. A valid schedule is an ordering of communication and processing events, subject to constraints including jitter and latency requirements, CPU and bus bandwidth constraints.

In addition to the number of processors and tasks, the most distinctive difference between this benchmark and the benchmark used in previous work[39] is the specification of data communications. In the old benchmark, data communications are specified between pairs of sender/receiver. However, in the new benchmark, the senders/receivers are decoupled by data items. Each process lists the data items it produces and consumes. Consequently, a datum may involve several sending processes and receiving processes, which usually translates into intricate constraints on the senders and receivers. The original tool fails to handle this type of constraints.

3.4.1 Modeling the problem

There are 6 processors, 1 SAFEbus, 52 application tasks, and 926 data items in the first cabinet. There are 6 processors, 1 SAFEbus, 48 application tasks, and 652 data items in the second cabinet. As there is no interaction between the two cabinets, we can partition the original problem into two subproblems.

In order to solve the AIMS problem, we wrote a translator to convert the specification in LISP format into the RTL-based input.

Each central processing module is modeled as a processor and each

process is instantiated to a sequence of jobs. A datum is also a process, whose frequency is the minimum rate of its sending and receiving processes. If a datum comes from outside the cabinet, its rate is equal to that of the receiving process.

In the input/output specification, a process describes the data items it produces and consumes. Since there can be several instances of a process and datum in a major frame, we need to determine the actual process instance that sends/receives a datum instance. Assume the rates of the datum (D), the sender(P_s) and the receiver(P_r) are f_d, f_s, f_r, respectively. For acyclic communications, we have

1. $f_s \leq f_r$

   Let $ratio_{sd} = f_s/f_d$ and $ratio_{rs} = f_r/f_s$.

   For datum $D : i$, the senders are chosen from the set $[P_s : (i \times ratio_{sd}), \ldots, P_s : ((i+1) \times ratio_{sd} - 1)]$. The possible receiver corresponding to the sender $P_s : j$ is from the set $[P_r : (j * ratio_{rs}), \ldots, P_r : ((j+1) * ratio_{rs} - 1)]$.

2. $f_s > f_r$

   Let $ratio_{rd} = f_r/f_d$ and $ratio_{sr} = f_s/f_r$.

   For datum $D : i$, the receivers are chosen from the set $[P_r : (i \times ratio_{rd}), \ldots, P_r : ((i+1) \times ratio_{rd} - 1)]$. The possible sender corresponding to the receiver $P_s : j$ is from the set $[P_s : (j * ratio_{sr}), \ldots, P_s : ((j+1) * ratio_{sr} - 1)]$.

For cyclic communications, we assume the higher rate process initiates the

communication first. For example, if there are cyclic communications between task A (2HZ) and B (4HZ), then a datum D0(1Hz) from A to B will be interpreted as

$$\begin{aligned} & A:0 \xrightarrow{D:0} B:1 \\ \vee \quad & A:1 \xrightarrow{D:0} B:3 \end{aligned}$$

A datum D1(1Hz) from B to A will be interpreted as

$$\begin{aligned} & B:0 \xrightarrow{D:0} A:0 \\ \vee \quad & B:2 \xrightarrow{D:0} A:1 \end{aligned}$$

After the translation process described above, there are 2336 jobs in the first cabinet. There are 10853 constraints imposed on these 2336 task instances. In the second cabinet, there are 1280 jobs and 6024 constraints.

3.4.2 Experimental Results

The new version of the MSP.RTL tool is written in Java. We evaluated the new tool with a range of scheduling problems on a Dell Precision desktop (Pentium 4 2.52GHz, 512M):

- Program 1 in Chapter 2.
- A simplified version of the AIMS problem from [39]. This is a single processor scheduling problem. The timing constraints include release time, deadline and jitter constraints.
- Another simplified version of the AIMS problem from [39]. In this problem, all communications among processes are taken out. This is a mul-

	number of task instances	*number of constraints*	*MSP.RTL running time (seconds)*
Program 1	4	18	0.03
Simplified example with all of the tasks running on processor P5 [39]	58	270	0.09
Simplified AIMS with no communication tasks [39]	458	2092	0.26
AIMS:			
cabinet 1	2336	10853	11.42
cabinet 2	1280	6024	unsolvable

Table 3.1: Benchmark results for the new MSP.RTL tool

tiprocessor scheduling problem. The timing constraints include release time, deadline and jitter constraints.

- Full version of the new benchmark problem. Cabinet 1 has 673 disjunctive constraints, each of which has up to 8 clauses. Cabinet 2 has 428 disjunctive constraints, each of which has up to 28 clauses.

Table 3.1 summarizes the results. Note the constraints in cabinet 2 are unsolvable according to the tool. We confirmed with Honeywell that the dataset in cabinet 2 has to been solved by preemptive scheduling.

Chapter 4

Applying MSP.RTL to Industrial Process Control

As an off-line multiprocessor scheduling tool, MSP.RTL is a good fit to industrial process control, where the networks are usually well designed and relatively fixed. As a matter of fact, we have used the tool to solve the block assignment problem in distributed Foundation Fieldbus systems. Recently, wireless process control has received a lot of attention. We investigated the feasibility of applying the tool to wireless industrial control systems. Preliminary results from both experiments are very promising.

Common to these two applications is that we need to develop a translator to transform the specific application requirements to RTL formulas. In addition, based on the characteristics of the application, we need to implement domain-specific heuristics in the MSP.RTL tool.

4.1 Optimizing Distributed Foundation Fieldbus Process Control

In a Foundation Fieldbus-based process control system, control strategies are implemented in a distributed environment on devices connected to

the fieldbus[50]. A control strategy is a policy to instruct a process to reach desired states, such as maintaining a tank level at a specified range. To implement a strategy, Foundation Fieldbus defines various building blocks which are known as function blocks in the specification, such as input blocks, output blocks and control blocks (implementations of control algorithms). A basic process control unit consists of some input blocks, a control algorithm, and some outputs. Periodically, the control algorithm reads the inputs, performs some calculation, writes to the outputs, and sometimes receives feedback from the outputs. These steps constitute a control loop. A control strategy is implemented by several control loops. The devices together execute the control strategy with a period called a *macrocycle*. Each control loop in the strategy can run several times in a *macrocycle*.

Given the requirement to control a process, the control engineer decides what process parameters to measure by sensor devices and what variables to control by actuator devices. The engineer then designs the control strategy by specifying control loops and their inter-connection. In addition to designing control strategies, a system engineer also needs to assign blocks to devices, normally done manually. Input blocks are allocated to sensors, output blocks are allocated to actuators. However, the assignment of control blocks is not trivial. Running control blocks on different devices can result in drastically different performance for the control strategy. We propose to use the MSP.RTL tool to solve the block assignment problem optimally.

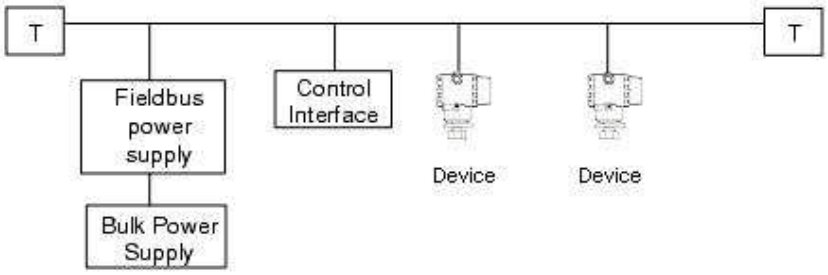


Figure 4.1: A Foundation FieldBus Network

4.1.1 Introduction to Foundation FieldBus Standard

Figure 4.1 is a sample fieldbus network. A device draws $4 - 20mA$ current from the network. The two terminators (T) at both ends can help prevent distortions and signal losses. The network communication is controlled by a master device which keeps the communications in order by sending out command messages. To execute a control strategy, a master device runs a schedule called LAS (Link Active Schedule). The LAS is executed repeatedly with a period called "macrocycle."

Owing to power drainage, the number of devices connected to a single Fieldbus network is limited. For example, a DeltaV [21] process control system allows at most 16 devices per network. In many practical situations, the number of actually installed devices is smaller than half of the limit (8).

Generally, a fieldbus device has certain capacity and can support several function blocks. Table 4.1 lists the blocks supported by three fieldbus devices. All blocks are standardized. For example, the PID blocks supported

Table 4.1: Fieldbus Devices and Function Blocks Supported

Manufacturer	Emerson Process Management	Siemens AG	Yokogawa Electric Corporation
Type	Temperature Transmitter	Valve Positioner	Vortex Flowmeter
Name	Rosemount 3244MV	IPART PS2 FF	YEWFLO W
Revision	4	1	2
Master device capable	yes	no	yes
AI count	3	1	2
AI time	50ms	N/A	100ms
AO count	0	1	0
AO time	N/A	50ms	N/A
PID count	2	1	1
PID time	100ms	75ms	160ms
Other blocks	AR, IS, SC	None	None

Block count indicates how many blocks of that type are supported in the device; Block execution time is measured in milliseconds (ms); AI: Analog Input; AO: Analog Output; PID: Proportional-Integral-Derivative Control; AR: Arithmetic; IS: Input Selector; SC: Signal Characterizer

by Rosemount 3244MV and *IPART PS2 FF* should have the same numbers of inputs and outputs, and they are required to implement the same PID algorithm. However, a block may have different execution times on different devices, as seen in Table 4.1.

The control interface in Figure 4.1 is usually installed as the gateway to the host process control system. Through this interface the user can configure and monitor the network.

Control strategies are distributed among the devices. With a common

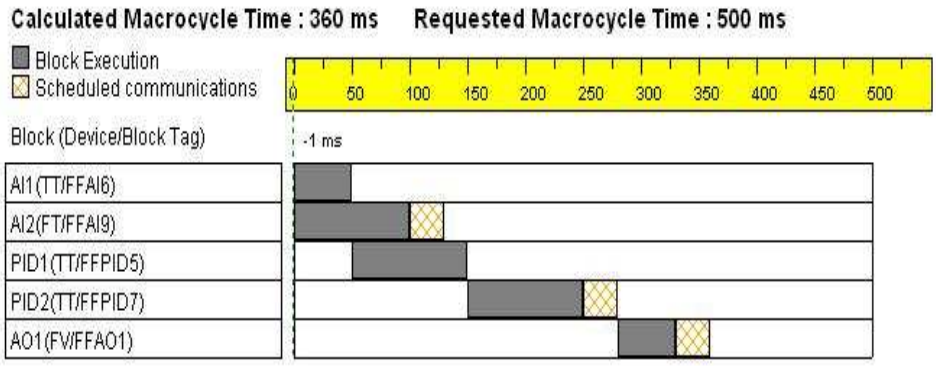


Figure 4.2: A FieldBus Macrocycle Schedule

clock, the devices together execute the control strategy at the rate of the macrocycle. We call the deployed control strategy on the fieldbus a *schedule*. The LAS in the master device controls when the synchronized input and output are transmitted. A function block in a device is pre-configured to run at certain time instants within the macrocycle to ensure that it is executed after its inputs are available and before its output are put into the sending buffers. Figure 4.2 is an example of a control strategy deployment in one macrocycle. The macrocycle is set at 500ms. The total schedule finishes in 360ms.

4.1.2 Schedule Generation Analysis

The Foundation Fieldbus standard specifies how a schedule is executed on the network, but leaves out the details about the generation of a schedule. Given the requirement to control a process, the control engineer decides what process parameters to measure by sensor devices and what variables to control

by actuator devices. The engineer then designs the control strategy by drawing function blocks and their inter-connections. Figure 4.3 shows a cascade control strategy, which is implemented by two control loops. A control loop is a basic control unit that consists of a control algorithm (implemented in one or more control blocks) with some inputs and outputs. In the outer loop of Figure 4.3, the control block PID1's input is set to be the AI1 block. The inner loop is controlled by the PID2 block, which reads input from the AI2 block and sends the output to the AO1 block. Eventually, AO1 feeds data back to PID2. Once the control engineer assigns blocks to devices, a schedule could be generated accordingly. In Figure 4.3, AI1, PID1, and PID2 are assigned to device TT; AI2 is assigned to device FT; AO1 is assigned to device FV. Assuming the models of devices TT, FT and FV are Rosemount 3244MV, YEWFLOW, and IPART PS2 FF from Table 4.1 respectively, the resulting schedule for the control strategy in Figure 4.3 is shown in Figure 4.2.

Several factors have to be considered in the process of deploying a control strategy:

1. *Block Assignment*: While input and output blocks have to be assigned to devices that sense or actuate the process, other blocks may be assigned to any device that supports them. In Figure 4.3 the blocks are pre-assigned. Several factors have to be taken into account when assigning blocks. First of all, a block can only be assigned to devices that support the block and have available resources to run it. For example, we may assign both PID blocks in Figure 3 to device TT because *Rosemount*

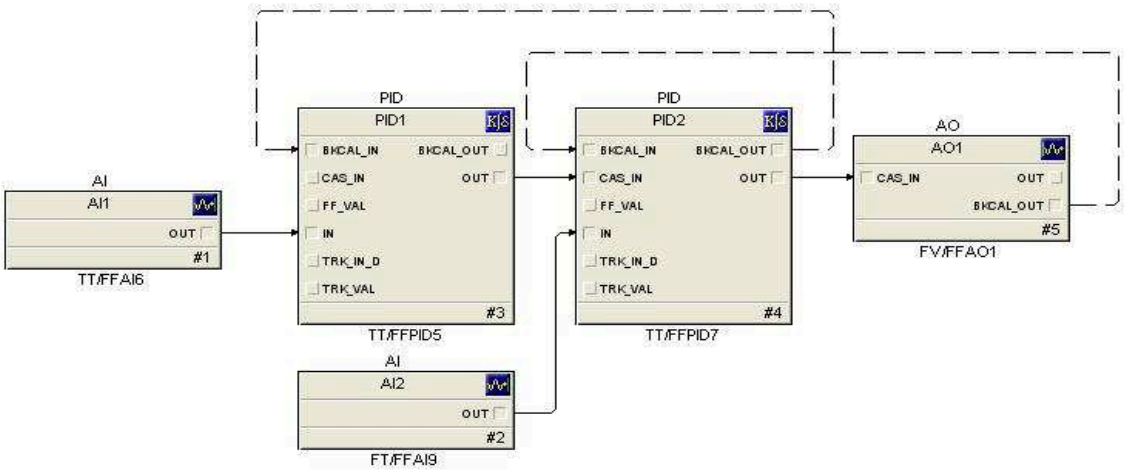


Figure 4.3: A Cascade Control Strategy

3244MV supports two PID blocks. Secondly, different devices may run a block at different speeds. Normally a faster device is preferred in order to reduce the total loop execution time. Thirdly, blocks communicating with one another should be ideally assigned to the same device so that the communication delay on the bus can be saved.

2. *Schedule One Control Loop*: As the control algorithms assume that the input values are periodically sampled, a control loop should start periodically and complete quickly in every period. Quick completion enables fast response to process variations. To reduce completion time, there should be no delays between consecutive block executions.

   As the control algorithms assume that the input values are periodically

sampled, running periodically can eliminate input jitters. This factor would not be a concern if the control strategy runs once every macrocycle. If a loop runs more than once within a macrocycle, each occurrence in the macrocycle must be placed so that over time the block's consecutive runs are evenly spaced.

3. *Schedule More Than One Control Loop*: The problem described in 2) is very simple if there is only one loop in a macrocycle. With more loops, we have to make sure that only one loop can publish data on the network at a time and a device can only publish data or run one block at certain time point. For example, in Figure 4.2, the device TT runs PID1 of the outer loop at time 50ms and PID2 of the inner loop at time 150ms. The schedule in Figure 4.2 can be further improved. The device FT can start AI2 at time 20ms instead of 0ms, and publish its data at time 70ms which is early enough for PID2 to pick up the value for its calculation. We may also schedule loops one after another. However, this has two drawbacks. For one thing, it unnecessarily increases the macrocycle length; we know that better control can be achieved with shorter loop execution period. For another thing, some loops are interdependent, in which case they have to be scheduled concurrently. In Figure 4.3, PID2 in the inner loop must be executed after PID1 in the outer loop. With these constraints, there are still many alternatives as to when a block or communication should be scheduled. In general, this is a complex scheduling problem that may not be solvable in polynomial time with respect to the number

of devices and blocks.

4. *The Control Interface*: In each fieldbus network, there is usually a control interface, which normally has no limit on how many blocks of one type can be run but does impose a limit on the number of total blocks.

5. *Macrocycle Determination*: If we can find the optimal schedule in terms of minimizing total execution time, we may be able to fit the control strategy in a shorter macrocycle, which means better control over the process. Thus we may want to derive the shortest macrocycle while preserving the schedule qualities mentioned above. In this research, we shall assume a given macrocycle in our experiment and leave the general problem for future work.

We now describe the problem formally. The notation $\mathbb{Z}_{\geq 0}$ denotes the set of non-negative integers. Formally, a control strategy C is a tuple $<T, \boldsymbol{D}, \boldsymbol{B}, \boldsymbol{L}, cap, ext>$ where:

- T is the macrocycle, the period of the control strategy
- $\boldsymbol{D}$ is a set of m devices $\{D_1, \ldots, D_m\}$
- $\boldsymbol{B}$ is a set of n function blocks $\{B_1, \ldots, B_n\}$
- $\boldsymbol{L}$ is a set of l control loops $\{L_1, \ldots, L_l\}$. Loop L_i is comprised of a sequence of q_i function blocks $\{B(i,1), \ldots, B(i,q_i)\}$, where $B(i,j) \in B$ for j, $1 \leq j \leq q_i$. If $B(i,j)$ and $B(i,j+1)$ run on different devices,

$B(i,j)$ sends its output to $B(i,j+1)$ over the shared bus. Otherwise the message would not appear on the bus. Communication delay over a bus is a constant E_{comm}. The period of loop L_i is T_i. $B(i,j,k)$ denotes the j-th block of the k-instance of loop L_i.

- *cap* is the *capability function*. $cap : \boldsymbol{B} \times \boldsymbol{D} \rightarrow \mathbb{Z}_{\geq 0}$. $cap(B_i, D_j)$ denotes the number of copies of blocks B_i allowed in device D_j. A value of 0 indicates device D_j does not support block B_i.

- *ext* is the *runtime function*. $ext : \boldsymbol{B} \times \boldsymbol{D} \rightarrow \mathbb{Z}_{\geq 0}$. $ext(B_i, D_j)$ denotes the execution time of B_i on D_j.

A deployment of control strategy C is a tuple $<assgn, sched>$, where:

- *assgn* is the *allocation function*. $assgn : \boldsymbol{B} \rightarrow \boldsymbol{D}$, which is defined as follows.

  (i) Suppose $assgn(B(i,j)) = D_k$. Then block $B(i,j)$ is allocated to device D_k.

  (ii) The number of block B_i assigned to device D_j is no bigger than $cap(B_i, D_j)$.

- *sched* is the schedule of the control blocks: $sched : \boldsymbol{B} \times \mathbb{Z}_{\geq 0} \rightarrow \mathbb{Z}_{\geq 0}$:

  (i) For every block $B(i,j,k),\ 0 \leq sched(B(i,j,k)) < T, 1 \leq i \leq l, 1 \leq j \leq q_i, 1 \leq k \leq T/T_i$

(ii) [**Jitter Requirement**] For every block $B(i,j,k)$, $sched(B(i,j,k+1)) = sched(B(i,j,k)) + T_i, 1 \leq i \leq l, 1 \leq j \leq q_i, 1 \leq k \leq T/T_i$

(iii) [**No Delay Requirement**] Suppose $B(i,j)$ and $B(i,j+1)$ are two consecutive blocks in loop L_i and k, $1 \leq k \leq T/T_i$. If $assgn(B(i,j)) = assgn(B(i,j+1))$, then $sched(B(i,j+1,k)) = sched(B(i,j,k)) + ext(B(i,j), assgn(B(i,j)))$. Otherwise $sched(B(i,j+1,k)) = sched(B(i,j,k)) + ext(B(i,j), assgn(B(i,j))) + E_{comm}$.

The finish time (*makespan*) of a strategy is $\mathbf{max}\{sched(B(i,j,k)) + ext(B(i,j), assgn(B(i,j))) \mid 1 \leq i \leq l, 1 \leq j \leq q_i, 1 \leq k \leq T/Ti\}$. The set of data messages produced by loop L_i $MSG(L_i)$ is $\{B(i,j) \rightarrow B(i,j+1) \mid 0 \leq j \leq T/Ti - 1, assgn(B(i,j)) \neq assgn(B(i,j+1))\}$. The number of messages appearing on the bus in this control strategy is $\sum_{i:1...l} T/T_i \times \|MSG(L_i)\|$.

Given a control strategy, the system engineer may choose to find a deployment that minimizes either the finish time of the strategy or the number of messages on the bus.

4.1.3 The Assignment Optimization Algorithm

As discussed in the subsection above, there should be no delay between consecutive blocks in a control loop. Based on this requirement, we add a new heuristics to the MSP.RTL tool. When the first block in a loop is scheduled, the start times of subsequent blocks in the loop can also be determined. MSP.RTL checks if there is any inconsistency in the potential schedule for subsequent blocks and reacts accordingly.

We now describe our algorithm for the block assignment problem. The basic algorithm is very straightforward and works as follows:

1. Find all possible combinations of assigning blocks to devices on the network. A block can be assigned to any device that supports blocks of that type. A device can only be assigned as many blocks as it supports.

2. For each combination, apply the MSP.RTL tool to produce one schedule candidate.

3. Select among all candidates the best schedules in terms of overall finish time and number of published data on bus.

4. Output the optimal schedules and associated deployments.

This procedure traverses the possible configuration space (a tree with a dummy root node) by a depth-first search. Each leaf node corresponds to a complete configuration. The output of this algorithm is the optimal configuration(s) as measured by certain metrics specified by the user.

Suppose the number of devices on the network is n, and the number of unassigned blocks is m. In the worst case where every block can be assigned to any device, the complexity of this algorithm is $O(n^m)$ which is exponential in n. However, the problem is often tractable in practice. The number of devices on a fieldbus network segment is limited in practice, and the number of unassigned control blocks within a network is expected to be small. We expect our algorithm to be able to solve a practical network configuration in seconds

or minutes, and this is not a big concern as the schedule is calculated offline during the configuration phase. In addition, certain deployment practices can further reduce the scheduling time. One example of such practice is the deployment of a control block. Oftentimes, a control block is assigned to one of the devices to which its input/output blocks are bound, or to the control interface. Note that input/output blocks are usually statically assigned to devices. By adopting this rule, the complexity of the algorithm above can be decreased to $O((k+1)m)$, where k is the maximum number of input/output devices for a control block (usually a small number). We call this assignment rule *Restricted-Control-Blocks (RCB)*.

It should also be noted that a simple branch-and-bound search technique may not help in the search process. Suppose the goal is to derive the earliest finish time and at a certain step we have a partial deployment (some blocks are left unassigned yet). It is hard to get a good estimate of the finish time for a partial deployment. During the next search step, another unassigned block is allocated to a device. From the perspective of real-time scheduling, some new tasks are added to the current task set. However, it is well known that in the context of multiprocessor non-preemptive real-time scheduling, adding a new task may in some cases reduce the makespan of the whole task set. It would be wrong to prune a search tree based on the current estimation on a partial deployment.

Finally, the translator transforms each block configuration as input to the MSP.RTL tool. In this step, the translator enforces the following require-

ments:

- All loops run strictly periodically, which means the maximum jitter of each process is 0.
- There is no delay between consecutive blocks in a loop. Suppose A and B are two such blocks, then we have $\downarrow A \leq \uparrow B \ \wedge \ \uparrow B \leq \downarrow A$.

We implemented the basic algorithm and the RCB heuristic in a Java program. The input to the program is a set of control strategies to be deployed on the network. Table II shows an example input file. Most sections of the file are self-descriptive. LANTENCY_COMM is the delay to publish data on the fieldbus. For fieldbus this parameter is configurable and is set at 30ms in our experiment.

All devices are listed in the NODES section. A device has three attributes: name, type and device model. The capability of each device model, as shown in Table I, is defined in an XML file. All blocks on the network are described in the BLOCKS section. Each block has three properties: name, type and the device it is assigned to. If the device field of a block is "?", the block is currently not assigned to any device. In practice, all input and output blocks are pre-assigned to devices. Only some control blocks need to be assigned by our program. The LOOPS section defines control loops on the network. For example, the loop AI363 $\rightarrow$ AI14002OUT contains block AI363 and AI14002OUT. AI363 sends its output to AI14002OUT. If the sending and receiving blocks are on the same device, the communication will not appear

Table 4.2: Configuration of One FieldBus Network

```
NETWORK      PlantA   # define network name
LATENCY_COMM 30       #time for data publish

NODES   # this section defines all devices in the network
    #  name          type        model name
    AT14002   actuator  2000Rev2
    LV14056   actuator  IPARTPS2FF
    FT15004   sensor    YEWFLOW
    TT15018   sensor    YEWFLOW

BLOCKS               # this section defines all function blocks
 # name         type   device
    AI363       AI     AT14002
    AI364       AI     AT14002
    AI367       AI     AT14002
    PID244      PID    ?
    PID247      PID    ?
    AO28        AO     LV14056
    AO34        AO     FV15004
    AI14002OUT  AO     PDT14059
    TI14002OUT  AO     AT14002

LOOPS                 # this section defines all control loops
    NAME AI14002
    PERIOD 1000
    LP_COMMS AI363 -> AI14002OUT
    END AI14002

    NAME TI14002
    PERIOD 1000
    LP_COMMS AI364 -> TI14002OUT
    END TI14002

    NAME LC14056
    PERIOD 1000
    LP_COMMS AI367 -> PID244 PID244 -> AO28 AO28 -> PID244
    END LC14056
```

on the bus and hence the delay would be 0ms. Each control loop runs at its own rate.

When a configuration as shown in Table II is converted to an input file to MSP.RTL, each device is represented by a processor. Blocks are mapped to processes running on processors. The execution time of each process is determined by the device it resides on. The period of a process is the period of the loop it belongs to. If a process is shared by several loops with different periods, a process runs with the shortest period.

The translation procedure must also conform to several criteria as discussed in Section 4.1.2:

1. All loops run strictly periodically, which means the max jitter of each process is 0.

2. There is no delay between consecutive blocks in a loop. This requirement can be translated into constraints in RTL format. For example, for the loop AI363 → AI14002OUT, the constraint is:
   $[@(\downarrow AI363) \leq @(\uparrow AI363 \rightarrow AI4002OUT)$
   $@(\uparrow AI363 \rightarrow AI4002OUT) \leq @(\downarrow AI363)$
   $@(\downarrow AI363 \rightarrow AI4002OUT) \leq @(\uparrow AI4002OUT)$
   $@(\uparrow AI4002OUT) \leq @(\downarrow AI363 \rightarrow AI4002OUT)]$

4.1.4 Experimental Results

To evaluate the performance of our algorithm, we use a real process control system configuration. The system has 78 fieldbus networks, more than 500 Fieldbus devices, and about 1500 input and output data points. As a fieldbus network can have at most 16 devices, the devices in this configuration are deployed on many separate fieldbus networks (on average fewer than 8 devices on a network).

The experiments are carried out on a Dell Precision 650 workstation, with Intel Xeon 2.4G CPU and 1GB memory. The operating system is Windows XP.

In the configuration above, all blocks are pre-assigned to devices and the load on each network is light. Our tool is able to calculate a schedule for this configuration within 1 second. The short running time can be attributed to the mutual independence of those fieldbus networks in this configuration. From the perspective of the MSP.RTL tool, the large scheduling problem is split into many unrelated simple sub-problems. Therefore, we shall focus on a single fieldbus network in following experiments.

To demonstrate the advantage of our tool, we create a heavy-loaded fieldbus network and compare our result with that produced by a commercial process control system. We select two networks in the configuration above and combine them into one network. The resulting network has 16 devices and 1 control interface. To be more specific, there are 12 input blocks, 12 output

blocks, and 5 PID blocks.

The program is first tested on the existing industrial deployment. Then the control blocks are marked "unassigned" incrementally: in the first round, only one control block is unassigned; in the final round, all 5 control blocks are unassigned.

The running time results are shown in Table 4.3. When only two control blocks are unassigned initially, the number of possible deployments is 133 and the basic algorithm finishes in 6 seconds. However, when the number of unassigned control blocks is up to 3, there would be 1753 possible assignments and the basic algorithm needs 78 seconds. It is obvious that the number of possible assignments and program response time increase exponentially with the number of unassigned control blocks.

Table 4.3 also shows the results with the RCB heuristic. The running time of RCB apparently increases sub-exponentially with the number of unassigned blocks. When all 5 PID blocks are unassigned, the basic algorithm need about 4.6 hours to try 384091 possible configurations, while it takes RCB only 8 seconds to try all 162 configurations.

We further examine the schedules generated by our tool. The details are shown in Table 4.4, where the first and last rows are from the commercial system and RCB, respectively. The other rows pertain to the basic algorithm. As the requirement of no delay between consecutive blocks within a loop is translated into constraints by this tool, the length of the total gaps within the

Table 4.3: Execution Time Vs Unassigned Blocks

Number of Unassigned PIDs	Number of Combinations		*Execution Time (seconds)*	
	Basic	RCB	Basic	RCB
0	1	1	< 1	< 1
1	11	3	< 1	< 1
2	133	9	6	< 1
3	1753	18	78	1
4	25013	54	1098	3
5	384091	162	16494	8

loops for all schedules by this tool is always zero, while that number for the commercial system is large (2070ms).

When all the blocks have been assigned to devices statically, the commercial system and the basic algorithm both finish within 1 second. However, the basic program produces a schedule with shorter overall finish time. By looking into the schedules, we find that the schedule produced by the basic program incurs less idle time on the bus. Based on Table 4.4, it is also obvious that for the basic algorithm, the schedule keeps improving with the increase in the number of unassigned blocks. However, the improvement is only on the schedule duration. This effect can be explained as follows. When more blocks become unassigned, each of those blocks can be allocated to the device that runs the block in the shortest time. This way, the duration of each loop can be reduced, which leads to the shorter schedule duration. On the other hand, a control loop may contain several input/output blocks and only one control block. As input and output blocks are pre-assigned to the devices, the number of data exchanged cannot be reduced much by allocating the control

Table 4.4: Schedules Generated by different methods

Deployment	Total Gaps within Loops(ms)	Schedule Duration	# of data publishing
the commercial system	2070	660	13
0 Unassigned, Fastest/IO	0	620	13
1 Unassigned, Fastest	0	555	12
1 Unassigned, Minimal I/O	0	555	12
2 Unassigned, Fastest	0	555	12
2 Unassigned, Minimal I/O	0	555	12
3 Unassigned, Fastest	0	530	13
3 Unassigned, Minimal I/O	0	555	12
4 Unassigned, Fastest	0	530	13
4 Unassigned, Minimal I/O	0	555	12
5 Unassigned, Fastest	0	505	13
5 Unassigned, Minimal I/O	0	555	12
with RCB	0	555	12

"Total gaps within loops" denotes the total unnecessary wait time between blocks within a loop. "Schedule duration" is the duration of the whole control strategy. "# of data publishing" is the number of data published on the network per macrocycle.

block to different devices. This explains the relatively constant value of data publishing in Table 4.4.

We also notice that the schedule derived by RCB is always the same for different numbers of unassigned blocks (from 1 to 5), and is at most as good as the corresponding schedule by the basic algorithm. This is because RCB restricts the devices a control bock can be assigned to, while the basic algorithm attempts to put a control block on every capable device. Basically, RCB trades the optimality of the schedule for the search speed.

Based on the results on the algorithm with RCB, we believe that our

tool is viable in industrial fieldbus deployment.

4.2 MSP.RTL in Wireless Process Control

Wireless process control has received a lot of attention in the past few years. In this research, we investigate the feasibility of applying wireless technologies in industrial real-time process control systems. There are several competing wireless process control standards, such as WirelessHART [27], ISA 100 [30], ZigBee [56]. In this part of the dissertation, we use ZigBee networks as the underlying wireless network protocol. We first define a minimum set of assumptions about the underlying wireless networks in order to achieve real-time support for process control. Then we formulate the modeling of a wireless process control configuration as a multi-processor scheduling problem, which is solved by the MSP.RTL tool. Simulation results on data from real plant configurations show that this approach can drastically reduce potential interferences between wireless transfers.

4.2.1 System Assumptions

A wireless process control system works just like a wired control system. A system engineer designs the control strategy, assigns blocks to devices, and designates the start time for each block. However, transmissions in wireless process control systems are unreliable. We must take into account transmission failures.

We focus on wireless process control systems built on top of ZigBee-

like networks. ZigBee is targeted at low-power and low data rate (250kb/s) communication environments. There are three types of devices in a ZigBee network: controllers, routers and end devices. A controller can be a router, whereas an end device cannot be a router. The maximum communication range for a ZigBee network can be as long as 100 meters. Currently, ZigBee devices can have about 100 days of battery life.

In order to achieve real-time control, we make three assumptions about the underlying wireless networks.

1. There is a way to program the sensors/actuators, either through some programming ports or wireless links (over-the-air-programming). As the schedule is generated offline by MSP.RTL, the devices have to be programmed with the schedule result.

2. The clocks of all devices in a network are synchronized. Several synchronization protocols for sensor networks [24, 35] can be readily applied to process control systems.

3. There is an upper limit on failed retries, i.e., the data transmission is assumed to eventually succeed after limited retries. Without a limit we can never achieve real-time control, even for hardwired systems. If the limit is exceeded, recovery action must be taken.

Assumption 1 is essential for the network to provide real-time services and is widely enforced in wireless sensor networks, as people always want to

get a meaningful timestamp on each triggered event. The development kit coming with ZigBee devices provides the capability stated in assumption 2. Assumption 3 is also reasonable because modern wireless technologies such as spread spectrum and adaptive radio could handle interferences fairly efficiently.

Based on assumption 3, only transient interferences are considered in this research. We cope with transient link failures in two ways. First, two communications which share an end node will be scheduled in non-overlapping time periods. This arrangement would avoid most hidden terminal problems, a major cause of interferences in a wireless network. Secondly, the scheduling tool takes into account possible retransmissions due to interferences. A transmission is allowed to retry a few times, in hope of recovering from temporary noise spikes. Consequently, real-time requirements can still be met as long as the number of retransmissions is within a limit.

4.2.2 Modeling a wireless process control system

We consider a wireless process control system built on top of a ZigBee mesh network. Sensors/Actuators are end devices. When a sensor cannot reach the controller directly, some intermediate routers have to be installed en-route to facilitate the communication. End devices cannot relay data from other devices. In addition, there can only be one controller in a network. These two requirements simplify our model: all control blocks will run in the controller or end devices; all input/output blocks will run in end devices (sensors/actuators). The sole function of the routers is to forward data packets.

Each function block can be mapped into a task with the corresponding period and run time. To schedule these tasks, a macro-cycle is first computed, during which each function block runs at least once.

Each device can be modeled as a processor. A router runs some type of routing algorithm. As the routing algorithm is not a periodic task, no periodic task is assigned to the routing process. Instead, a routing delay is used to simulate the routing process. Each function block can be mapped into a task with the corresponding period and run time. We assume the communication path between two blocks is fixed and specified by the system engineer.

Until this point, this scheduling problem can be regarded as several independent scheduling problems on multiple processors. However, several aspects of wireless process control systems make this problem nontrivial:

1. Communications between function blocks. Communications introduce many intricacies to this problem. First, two blocks that exchange data may have different periods if they belong to different loops. Secondly, a node cannot receive packets from multiple sources simultaneously. This is also the case when a node feeds the inputs to several receivers. Thirdly, there may be interferences between two communications. Communications are covered in Section 4.2.2.1.

2. Two control loops may share one function block. In addition, the two control loops may have different periods. So a loop might only run certain instances of a block.

3. A router may be shared by several control loops. A single control loop may traverse a router twice. In order to avoid interferences, the scheduler should assign the communications through that router to non-overlapping time slots.

4.2.2.1 Communications

A usual practice in real-time scheduling is to model communications as processes running on a bus. In a wireless control system, not all devices share a single bus. Two communications may take place concurrently if the transmitter and receiver of one communication are separated far enough from those of the other. To capture this property of wireless communications, multiple buses are created for a wireless control system. In our model, all communications in a control loop are considered running on a dedicated bus. As blocks in a control loop run sequentially in time, there would be no competition for the bus in a loop. We only need to consider the interferences between loops.

As routing processes are represented by routing delays, the two end points of a communication must be function blocks. For example, if a control block (PID1) sends a packet to an output block (AO1) through the router (R2), the communication is represented as PID1→ R2→AO1.

If the two end blocks of a communication run at different rates, it can only be the case that the source block runs at a lower rate than the destination block, otherwise the receiver would be overrun. In this case, the source block can only send messages to certain instances of the destination block.

4.2.2.2 Control Loops

A control loop is comprised of block processes and communications that have to run in a sequential order. For example, in the control loop (AI1, AI1 → PID1, PID1, PID1 → AO1, AO1), AI1→PID1 can only start after the termination of AI1, and PID1 cannot start until the end of AI1→PID1.

Note that Some control loops contain feedbacks.

4.2.2.3 Shared routers and blocks

Routers and blocks may be shared by multiple control loops. In order to avoid interferences, we want to make the communications through shared routers/blocks run in non-overlapping time slots. Mutual exclusion can be easily described by an OR constraint in RTL. For example, if process A and B cannot run at the same time, this requirement can be expressed as:

$$\downarrow A \leq \uparrow B \vee \downarrow B \leq \uparrow A$$

Two communications without shared end nodes (blocks/routers) can still interfere with each other. However, the probability of interference is reduced dramatically. To address those infrequent interferences, we allow a communication process to retry a fixed number of times. We expect that the probability of transmission failures with several retries should be negligibly low and acceptable. In case such failure does happen, we can always resort to some emergency reporting mechanisms (such as alarms). It is the same case with traditional wireline control systems.

Formally, we can define the problem as follows. A control strategy on a ZigBee-based network configuration is a tuple $< T, D, R, B, L, assgn >$ where:

- T is the macrocycle, the period of the control strategy
- D is a set of devices including end devices and the controller
- R is a set of routers
- B is a set of blocks
- L is a set of control loops defined over B. Each loop L_i with period T_i is a sequence of blocks $\{B(i,1), \ldots, B(i,q_i)\}$. The communication path $P(i,j)$ between two consecutive blocks $(B(i,j), B(i,j+1))$ in loop L_i is the two end blocks connected by a sequence of routers $\{B(i,j), R(i,j,1), \ldots, R(i,j,p_{ij}), B(i,j+1)\}$. $B(i,j) \rightarrow B(i,j+1)$ denotes a message from $B(i,j)$ to $B(i,j+1)$ following the path $P(i,j)$.
- $assgn$ is the *allocation function*. $assgn : B \rightarrow D$. Block $B(i,j)$ in loop L_i is assigned to device $assgn(B(i,j))$.

Recall that $\uparrow A$ denotes the start time of action A and $\downarrow A$ denotes the completion time of action A. Given such a configuration, we want to find a schedule S for the wireless configuration such that:

1. $0 \leq \downarrow B(i,j,k) < T$ for each block $B(i,j)$ in loop L_i and $1 \leq k \leq T/T_i$.
2. $\downarrow(B(i,j) \rightarrow B(i,j+1)) \leq \uparrow(B(k,l) \rightarrow B(k,l+1)) \vee \downarrow(B(k,l) \rightarrow B(k,l+1)) \leq \uparrow(B(i,j) \rightarrow B(i,j+1))$ if $P(i,j) \cap P(k,l) \neq \emptyset$.

Table 4.5: Summary of the real plant configuration

FieldBus Networks	25
Nodes(including controllers and devices)	158
Control loops	125
Function blocks	311
Communications	202

4.2.3 Simulation Results

In order to evaluate our approach, we use as benchmark a real plant configuration based on Foundation Fieldbus. In our experiments, this configuration is first converted into a ZigBee-based process control system. Then, the resulting wireless control configuration is translated into a multiprocessor scheduling problem to be solved by MSP.RTL

4.2.3.1 The target wireless process control system

The summary for the real plant configuration is shown in Table 4.5.

In this configuration, all control loops have the same period, $1000ms$ (1 second). The average run time of a control block is $10ms$, and the average run time of an input/output block is $45ms$. The communication delay is $30ms$. Based on the plant configuration on Fieldbus, we construct the corresponding wireless control systems in different topologies. ZigBee supports three types of network topologies: star, cluster and mesh. In a star topology, all devices communicate with the central controller directly, without any relaying routers. As all devices on a Fieldbus network share a single bus, there is an inherent similarity between a Fieldbus network and a ZigBee network in a star topol-

ogy. Thus we first map the configuration to a wireless control system based on ZigBee star networks. The mapping process is trivial: all definitions of the wireline system can be literally copied to the wireless system with the exception of communications. As mentioned in Section 4.2.2, we allow a wireless communication to retry for a limited number of times. Thus the number of maximum retransmissions is specified as a parameter in the definition of a wireless control system. Other than this parameter, there exists a one-to-one mapping between the Fieldbus plant configuration and the ZigBee configuration. The resulting wireless control system is saved in a data file which lists the devices in the network, the function blocks on each device and the composition of each control loop.

4.2.3.2 Experimental Results

In order to schedule control loops in the wireless control system, we wrote a parser to translate the configuration file to a scheduling problem definition in RTL format. The details of the translation process were discussed in Section 4.2.2.

The parser and MSP.RTL are written in Java with Netbeans IDE 5.0. The experiments are carried out on a Dell Latitude C400 laptop running Windows XP with Intel Pentium III 1.2GHZ CPU and 512MB memory. The run time is averaged over 10 runs of each program.

As the first step, the definition of the wireless control system is fed to the parser. The resulting problem has 283 processors, 512 processes, and 1190

Table 4.6: Schedule results for different retries

Max Retries	Schedulability	Run Time (s)
2	Yes	7
3	Yes	8
4	Yes	9
5	No	0

constraints. Of the 283 processors, 125 processors are virtual buses created for communications, which is more than the number of networks (25). For networks in a star topology, we could be better off creating only one bus for each network. For networks in cluster/mesh networks, there could be multiple buses associated with a network. In order to ensure that our parser works for networks in all three topologies, we choose to create a bus for each loop, as described in Section 4.2.2.

The run time of the parser is always less than 1.0 second, which is negligible for design purposes. In the second step, we run MSP.RTL on the output created by the parser. We vary the maximum number of retransmissions from 2 to 5. The result is shown in Table 4.6.

As shown in Table 4.6, the run time goes up with the increase in the maximum number of retries. When the number of retries is incremented, each communication takes a longer time, which causes MSP.RTL to allocate more time to each communication and thus to spend more time finding a feasible schedule. When the maximum retries reaches 5, MSP.RTL promptly reports that this problem is unschedulable. It is worth noting that the run time of MSP.RTL is more related to the internal structure of a wireless control system

than the size of the system. For MSP.RTL, if the size of the control system is increased by adding more networks to it, the running time is increased at most linearly. However, if the size of a system is kept intact and only one network becomes more complex, MSP.RTL may take much longer time to find a schedule. In an extreme case, if a network is unschedulable by MSP.RTL, the whole control system would be unschedulable.

To validate the schedule independently, we develop a verification tool which checks if the schedule is consistent with all constraints in the problem. The schedule created by MSP.RTL passes the verification successfully.

Chapter 5

WirelessHART: Building a Real Wireless Industrial Process Control Network

Wireless process control has been a popular topic recently in the field of industrial control [11, 12, 52]. Compared to traditional wired process control systems, their wireless counterparts have the potential to save costs and make installation easier. Also, wireless technologies open up the potential for new automation applications. Several industrial organizations, such as ISA [30], HART [27], WINA [54] and ZigBee [56], have been actively pushing the application of wireless technologies in industrial automation. As a milestone of such efforts, WirelessHART is ratified by the HART Communication Foundation in September 2007. WirelessHART is the first open wireless communication standard specifically designed for process measurement and control applications [27].

HART (Highway Addressable Remote Transducer) is a global standard for smart process instrumentation and the majority of smart field devices installed in plants worldwide are HART-enabled. It is estimated that the global installed base of HART-enabled devices is at more than 20 million [27]. WirelessHART provides host applications wireless access to existing HART-enabled

field devices and supports the deployment of battery operated, wireless-only HART-enabled field devices.

Before WirelessHART is released, there have been a few publicly available standards on office and manufacturing automation, such as ZigBee [56] and Bluetooth [2]. However, these technologies cannot meet the stringent requirements of industrial control. Compared with office applications, industrial applications have stricter timing requirement and higher security concern. For example, many monitoring applications are expected to retrieve updates from sensors every one second. Neither ZigBee nor Bluetooth makes any effort to provide a guarantee on end-to-end wireless communication delay. In addition, industrial environments are harsher for wireless applications in terms of interferences and obstacles than office environment. Some interference may be persistent. ZigBee, without built-in channel hopping technique, would surely fail in such environments. Bluetooth assumes quasi-static star network, which is not scalable enough to be used in large process control systems.

The new WirelessHART is specifically targeted to solve these problems and provide a complete solution for process control applications. At the very bottom, it adopts IEEE 802.15.4-2006 [4] as the physical layer. On top of that, WirelessHART defines its own time-synchronized MAC layer. Some notable features of WirelessHART MAC include strict $10ms$ time slot, network wide time synchronization, channel hopping, channel blacklisting, and industry-standard AES-128 ciphers and keys. The network layer supports self-organizing and self-healing mesh networking techniques. In this way, mes-

sages can be routed around interferences and obstacles. It is noteworthy that WirelessHART distinguishes itself from other public standards by maintaining a central network manager. The network manager is responsible for maintaining up-to-date routes and communication schedules for the network, thus guaranteeing the network performance.

In chapter 4, we make several assumptions on wireless process control systems and apply the MSP.RTL tool to schedule communications in a simulated wireless networks. As WirelessHART is released as an industry standard, we are interested in building a real wireless process control system based on WirelessHART and experiment with such a real system.

In this chapter we discuss how we developed a prototype WirelessHART protocol stack. Based on the prototype, we build a four-node network to demonstrate the join process of a new device and the execution of a simple control loop.

For practical concern, we need to implement the feature-rich WirelessHART on controllers with low processing power and limited resources. We identified and conquered some challenges, such as time management, communication security, and mesh networking. To the best of our knowledge, this effort is the first reported attempt to implement the newly approved WirelessHART standard. Those who want to build a full-featured WirelessHART stack should find our experiences helpful.

5.1 Background and Related works

Conceptually, WirelessHART networks are one special type of wireless sensor network. Although it bears many similarities with other wireless standards, such as Bluetooth [2], ZigBee [56], and Wi-Fi [3], WirelessHART differentiates itself from them in many other aspects.

Wireless sensor network has received extensive study recently [7, 8, 20, 34, 53, 55]. Different from generic wireless sensor networks which assume that sensors are deployed randomly and abundantly, the deployment of WirelessHART network is deliberate and has only limited redundancy. In a generic sensor network, many sensors may be deployed in the same area and perform the same function. However, in a WirelessHART network, sensors are usually attached to field devices to collect specific environmental data, such as flow speeds, fluid levels, or temperatures. A reading from a sensor is not necessarily replaceable by that from the nearby sensors. More important, generic wireless sensor networks are self-configurable and have no strict requirements on timing and communication reliability. To meet the requirements of wireless industrial applications, WirelessHART uses a central network manager to provide routing and communication schedules. From this point of view WirelessHART is essentially a centralized wireless network.

WirelessHART, Bluetooth and ZigBee share a very obvious feature: they all operate in the unrestricted 2.4GHz ISM radio band, which is available almost globally. On the other hand, they distinguish from each other in many other aspects. Both WirelessHART and Bluetooth support time slots and

channel hopping. However, Bluetooth is targeted at Personal Area Networks (PAN), whose range is usually set to 10 meters. Furthermore, Bluetooth only supports star-type network topology, and one master can only have up to 7 slaves. These limitations make it awkward to apply Bluetooth in large industrial control systems. In contrast, WirelessHART supports mesh networking directly. The topology of a WirelessHART network can be a star, a cluster or a mesh, thus providing much better scalability.

Both WirelessHART and ZigBee are based on the IEEE 802.15.4 physical layer. While ZigBee uses the existing IEEE 802.15.4 MAC, WirelessHART goes one step further to define its own MAC protocol. WirelessHART introduces channel hopping and channel blacklisting into the MAC layer, while ZigBee can only utilize Direct Sequence Spread Spectrum (DSSS) built in IEEE 802.15.4. Thus, if a noise is persistent, which is not unusual in industrial field, the performance of a ZigBee network might degrade severely. By changing the communication channel pseudo-randomly, WirelessHART can limit the damage to minimum.

Wi-Fi is based on the IEEE 802.11 standards and its spectrum assignments and operational limitations are not consistent worldwide. In addition, its power consumption is fairly high compared to other low-bandwidth standards like ZigBee, Bluetooth, and WirelessHART, which makes it not a good fit for industrial environment as well.

It is noteworthy that ISA SP100 [30] committee is also working on a wireless standard for industrial applications. The committee approved the

first ISA100 standard on September 9th, 2009. It is almost two years after the release of WirelessHART and its market acceptance is yet to see.

5.2 WirelessHART Architecture

Figure 5.1 illustrates the architecture of the WirelessHART protocol stack according to the OSI 7-layer communication model. As shown in this figure, WirelessHART protocol stack includes five layers: physical layer, data link layer [1], network layer, transport layer, and application layer. In addition, a central network manager [48] is introduced to manage the routing and arbitrate the communication schedule. That is, it creates routing table for each device and dictates which two nodes are able to communicate at a specific time slot on a certain channel.

For the ease of presentation, we define the terms and abbreviations used in the rest of this chapter in Table 5.1.

5.2.1 Physical layer

The WirelessHART physical layer is based mostly on the IEEE STD 802.15.4-2006 2.4GHz DSSS physical layer [4]. This layer defines radio characteristics, such as the signaling method, signal strength, and device sensitivity.

Just as IEEE 802.15.4 [4], WirelessHART operates in the 2400-2483.5MHz license-free ISM band with a data rate of up to 250 Kbits/s. Its channels are

[1] In the rest of this paper, we use "data link layer" and "MAC layer" interchangeably.

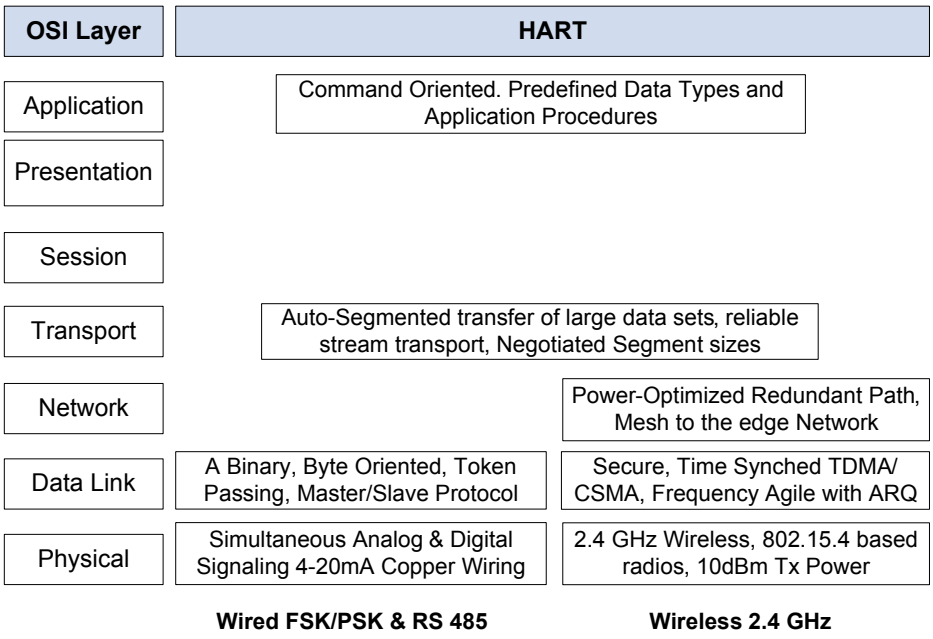


Figure 5.1: Architecture of HART Communication Protocol

Table 5.1: Definitions and Abbreviations.

Term	**Definition**
AES	Advanced Encryption Standard
ASN	Absolute Slot Number
BDM	Background Debug Module
CCA	Clear Channel Assessment
CCM*	Counter with CBC-MAC (corrected)
DLPDU	Data Link Protocol Data Unit
DSSS	Direct Sequence Spread Spectrum
EIRP	Equivalent Isotropic Radiated Power
MCU	Micro Control Unit
MIC	Message Integrity Code
NPDU	Network Protocol Data Unit
PIB	Protocol Information Base
SP	Service Primitive
TDMA	Time Division Multiple Access

numbered from 11 to 26, with a 5MHz gap between two adjacent channels.

To meet strict timing requirements, WirelessHART mandates that radio transceivers be compliant to the IEEE 802.15.4 standard and meet the following criteria: (1) the maximum switching time between channels shall be 12 symbol periods (0.192 ms); (2) the maximum radio turn-on time should be 4 ms; (3) the power level of the device must be controlled (programmable) at discrete, monotonic levels from -10 dBm to $+10$ dBm EIRP (with ± 2 dBm of error in actual power level); and (4) receiver sensitivity should be -90 dBm or better.

5.2.2 Data Link Layer

One distinct feature of WirelessHART is the time-synchronized data link layer. WirelessHART defines a strict $10ms$ time slot and utilizes TDMA technology to provide collision free and deterministic communications. The concept of *superframe* is introduced to group a sequence of consecutive time slots. Note a superframe is periodical, with the total length of member slots as the period. All superframes in a WirelessHART network start from the same ASN (absolution slot number) 0, the time when the network is first created. Each superframe then repeats itself along the time based on its period. For example, two superframes A and B, A has 3 slots and B has 4 slots. Then ASN 9 is the first slot in the 4th instance of superframe A, and the second slot in the 3rd instance of superframe B. A WirelessHART device must be able to support multiple superframes so that it can follow different communication

schedules upon the request of the network manager.

In WirelessHART, a transaction in a time slot is described by a vector: $\{frame\_id, index, type, src\_addr, dst\_addr, channel\_offset\}$, where $frame\_id$ identifies the specific superframe; $index$ is the index of the slot in the superframe; $type$ indicates the type of the slot (transmit/receive/idle); src_add and dst_addr are the addresses of the source device and destination device, respectively; $channel\_offset$ provides the logical channel to be used in the transaction.

To fine-tune the channel usage, WirelessHART introduces the idea of *channel blacklisting*. Channels affected by consistent interferences could be put in the black list. In this way, the network administrator can disable the use of those channels in the black list totally.

To support channel hopping, each device maintains an active channel table. Due to channel blacklisting, the table may have less than 16 entries. For a given slot and channel offset, the actual channel is determined from the formula:

ActualChannelIndex = (ChannelOffset + ASN) % NumChannels

Then *ActualChannelIndex* is used to index into the active channel table to get the physical channel number. Since the ASN is increasing constantly, the same channel offset may be mapped to different physical channels in different slots.

The strict time synchronization in WirelessHART also makes the channel hopping technology [10] practical. It allows the communicating devices

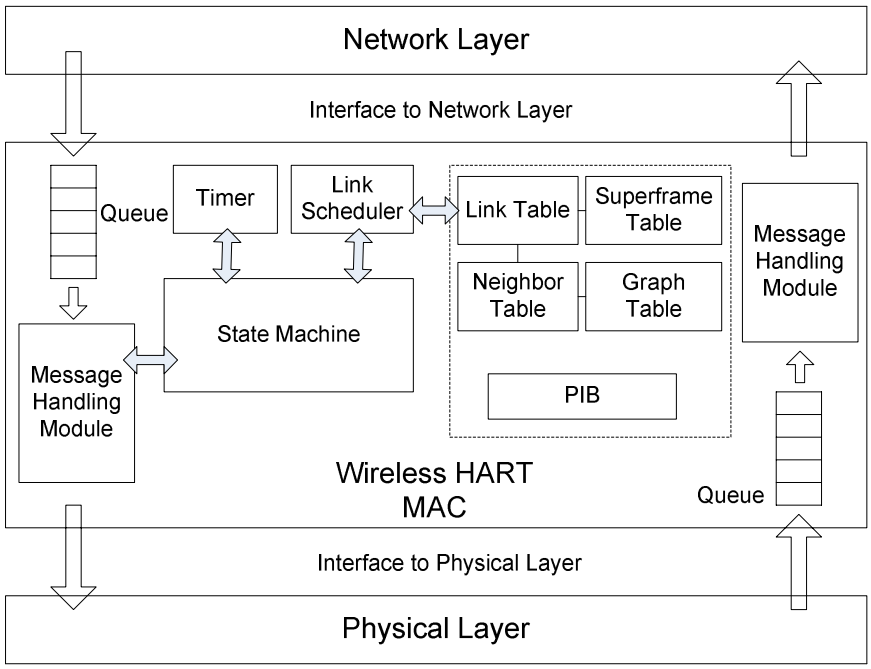


Figure 5.2: WirelessHART Data Link Layer Architecture

to rendezvous in hopping channels, thus providing frequency diversity and enhancing the communication reliability.

Figure 5.2 describes the overall design of the data link layer which consists of six major modules.

5.2.2.1 Interfaces

As the data link layer sits between the physical layer and the network layer, the interface among them has to be defined clearly. Basically, the interface between the MAC and PHY layer describes the service primitives provided

by the physical layer, and the interface between the MAC and NETWORK layer defines the service primitives provided to the network layer. This interface defines the MAC operations from a "black box" point of view.

5.2.2.2 Timer

Timer is a fundamental module in WirelessHART. It provides accurate timing to ensure the correct operating of the system. One significant challenge we met during the implementation is how to design the timer module and keep those $10ms$ time slots in synchronization. The specific timing requirement inside a WirelessHART time slot is depicted in Fig. 5.3. When a node wants to send a frame, it first does a clear-channel-assessment(CCA) check at $TsCCAOffset$ time units after the start of the time slot. If the channel is clear, it starts to transmit the frame at $TsTxOffset$. After finishing sending the frame, it switches the transceiver from Tx mode to Rx mode and waits for the acknowledgement. On the receiver side, the receiver waits $TsRxOffset$ time units to listen for frames. After receiving the end of the frame, it waits $TsTxAckDelay$ to send out the acknowledgement.

5.2.2.3 Communication Tables

Each network device maintains a collection of tables in the data link layer. The superframe table and link table store communication configurations created by the network manager; the neighbor table is a list of neighbor nodes that the device can reach directly and the graph table is used to collaborate

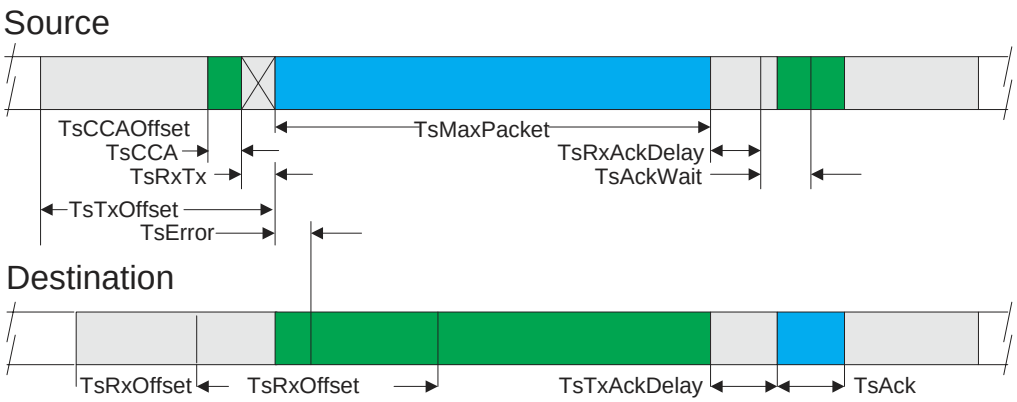


Figure 5.3: WirelessHART Slot Timing

with the network layer and record routing information.

In addition to these tables, a protocol information base (*PIB*) is created to keep track of the device's configuration parameters and current status. Various service primitives are provided by the interfaces to read and write these configuration information.

5.2.2.4 Link Scheduler

The functionality of the link scheduler is to determine the next slot to be serviced based on the communication schedule in the superframe table and link table. The scheduler is complicated by such factors as transaction priorities, the link changes, and the enabling and disabling of superframes. Every event that can affect link scheduling will cause the link schedule to be re-assessed.

The link scheduler first checks the DLPDUs in the outgoing queue and

determines the first absolute slot number (ASN) that can be used to transmit a DLPDU. Next, it iterates through all receive links to determine the first ASN for listening. The smaller of the two ASNs selected above will be scheduled for servicing. Ties are resolved in favor of transmit slots.

5.2.2.5 Message Handling Module

The message handling module buffers the packets from the network layer and physical layer separately. As WirelessHART defines four priorities for DLPDU's, a DLPDU with higher priority must be inserted before a DLPDU with lower priority. In addition, this module supports message service primitives that locate a packet by the packet handle, *e.g.*, FLUSH.request and FLUSH.confirm.

It coordinates with the state machine, security manager and the link scheduler to decide which packet for the specified link should be selected, encrypted/decrypted and forwarded to its destination.

5.2.2.6 State Machine

The state machine in the data link layer consists of three primary components: the TDMA state machine, the XMIT and RECV engines. The TDMA state machine is responsible for executing the transaction in a slot and adjusting the timer clock. The XMIT and RECV engine deal with the hardware directly, which send and receive a packet over the transceiver, respectively. After the link scheduler decides the next slot to be serviced, it

invokes the TDMA state machine, passing as parameters the transaction in the slot and the corresponding packet (if available). The TDMA machine handles the details of the transaction, such as the timing requirement in a time slot, calculating the message MIC(Message Integrity Code), sending/receving the DLPDU, and awaiting/sending the acknowledgement.

More information on the state machine design is presented in Section 5.3.4.

5.2.3 Network Layer and Transport Layer

WirelessHART supports a wide range of network topologies. The network layer and transport layer cooperate to provide secure and reliable end-to-end communication for network devices.

As shown in Figure. 5.4, the basic elements of a typical WirelessHART network include: (1) **Field Devices** that are attached to the plant process, (2) **Handheld** which is a portable WirelessHART-enabled computer used to configure devices, to run diagnostics, to perform calibrations, and to manage network information inside each device, (3) A **gateway** that connects host applications with field devices, and (4) A **network manager** that is responsible for configuring the network, scheduling and managing communication between WirelessHART devices. Commonly, the network manager is integrated in the Gateway.

Figure. 5.5 describes the detailed architecture of the network layer. For the purpose of security, WirelessHART is session oriented and a session

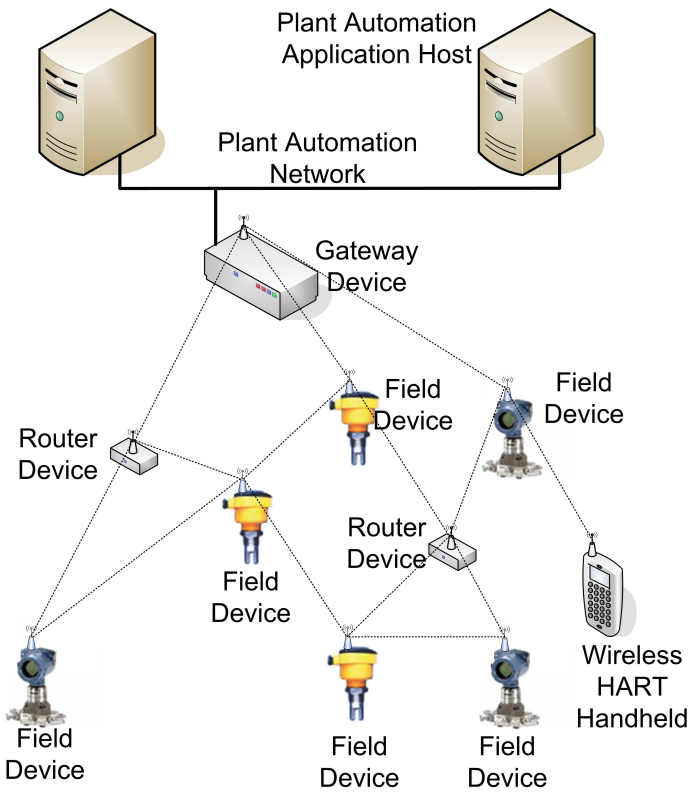


Figure 5.4: WirelessHART Mesh Networking

enables confidential and secure communication between two end nodes. Each NPDU will be enciphered before being sent out. At the destination device, the NPDU will be authenticated and deciphered. The details of the encryption, decryption and authentication mechanisms in WirelessHART will be described in Section 5.3.

The transport layer of WirelessHART supports the acknowledged operation which is used to construct a synchronous transport pipe across the network connecting devices. The transport pipe allows devices to send pack-

ets and to confirm their delivery in a fixed order.

To support the mesh communication technology, each WirelessHART device is required to be able to forward packets on behalf of other devices. There are two routing protocols defined in WirelessHART:

5.2.3.1 Graph Routing

A graph is a collection of paths that connect network nodes. The paths in each graph is explicitly created by the network manager and downloaded to each individual network device. To send a packet, the source device writes a specific graph ID (determined by the destination) in the network header. All network devices on the way to the destination must be pre-configured with graph information that specifies the neighbors to which the packets may be forwarded. In a properly configured network, all devices will have at least two devices in the graph through which they may send packets. This redundancy is necessary for network layer transmission retries.

5.2.3.2 Source Routing

Source Routing is a supplement to the graph routing aiming at network diagnostics. To send a packet to its destination, the source device includes in the header an ordered list of devices through which the packet must travel. As the packet is routed, each routing device utilizes the next network device address in the list to determine the next hop until the destination device is reached. Otherwise the device at the point of failure must notify the network

manager and discard the packet. It is the responsibility of the network manager to take corrective action.

In Section 5.3.5, we discuss mesh networking in more details and present our design of the underlying network data model.

5.2.4 Application Layer

The application layer is the topmost layer in the WirelessHART standard. It defines various commands, data types and status. Since it is command-oriented, communications between peers are represented as command requests and responses. The application layer is responsible for parsing the messages from its peer, extracting the command numbers, executing the specified commands, and generating responses.

The format of the command message is shown in Figure 5.6. The command number can be up to two bytes, which means as many as 65535 commands can be defined. The command number is followed by a one-byte *byte count* field and then the command payload. Currently, the WirelessHART specification has defined hundreds of commands. Although a WirelessHART device does not need to implement all of them, three classes of commands are mandatory for all devices.

- **Universal Commands** are used to provide the identification and software configuration information. For example, Command 0 is used to read the device's 5-byte unique ID.

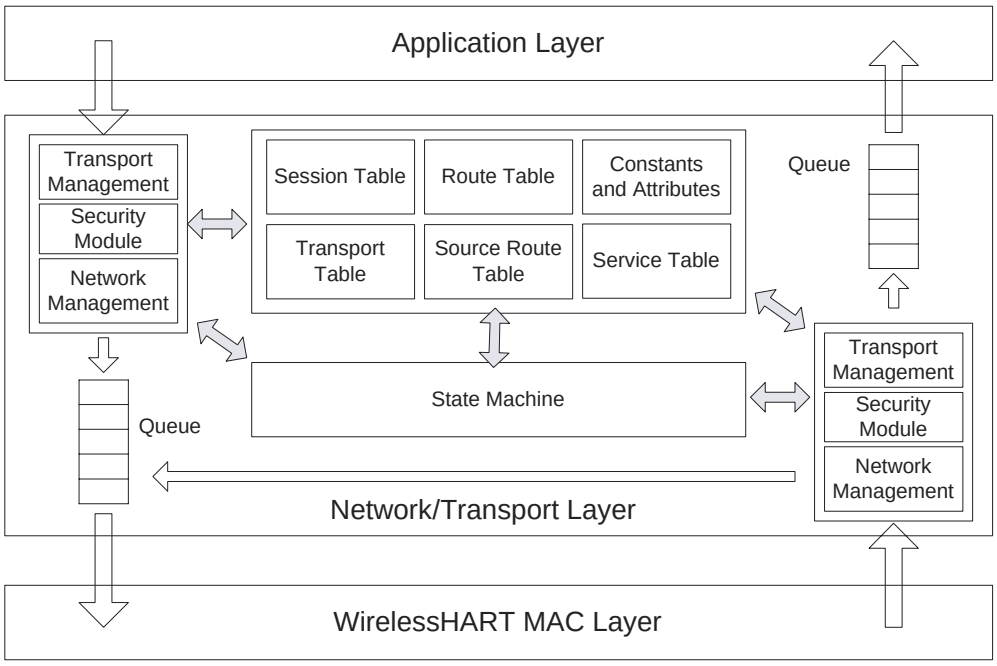


Figure 5.5: WirelessHART Network Layer Architecture

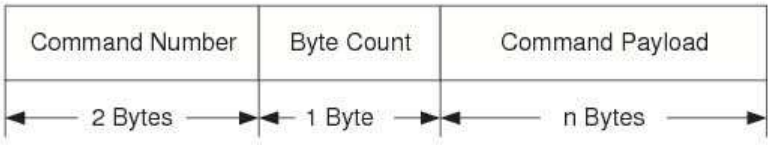


Figure 5.6: Command Message Format

- **Common Practice Commands** are mainly used to provide process data services like the burst data service.

- **Wireless Commands** are used to configure and maintain the WirelessHART network. For example, Command 965 is used to write superframes to the designated device.

Among these three classes, wireless commands are newly defined in the WirelessHART specification. These wireless commands wrap all services implemented by network layer, data link layer and physical layer, and provide unified interfaces for the wireless network management.

5.2.5 Security Architecture

WirelessHART is a secure network system. Both the MAC layer and network layer provide security services. The MAC layer provides hop-to-hop data integrity by using MIC. Both the sender and receiver use CCM* mode together with AES-128 as the underlying block cipher to generate and compare the MIC.

The network layer employs various keys to provide confidentiality and data integrity for end-to-end connections. Four types of keys are defined in the security architecture:

- **Public Keys** which are used to generate MICs on the MAC layer by the joining devices.
- **Network Keys** which are shared by all network devices and used by existing devices in the network to generate MAC MIC's.
- **Join Keys** that are unique to each network device and is used during the joining process to authenticate the joining device with the network manager.

- **Session Keys** that are generated by the network manager and is unique for each end-to-end connection between two network devices. It provides end-to-end confidentiality and data integrity.

Figure 5.7 describes the usage of these keys under two different scenarios: 1) a new network device joining the network, and 2) an existing network device communicating with the network manager. In the first scenario, the joining device will use the public key to generate the MIC on MAC layer and use the join key to generate the network layer MIC and encrypt the join request. After the joining device is authenticated, the network manager will create a session key for the device and thus establish a secure session between them. In the second scenario, on the MAC layer, the DLPDU is authenticated with the network key; on the network layer, the packet is authenticated and encrypted by the session key.

5.3 Challenges and Solutions

As described in Section 5.2, WirelessHART includes some core modules, such as time management, mesh networking, security and network management. It is a very challenging task to build such a prototype on a resource-limited hardware platform.

In the following subsections, we first introduce the hardware platform we use. Then we describe those challenges we met in the process of development and present our solutions.

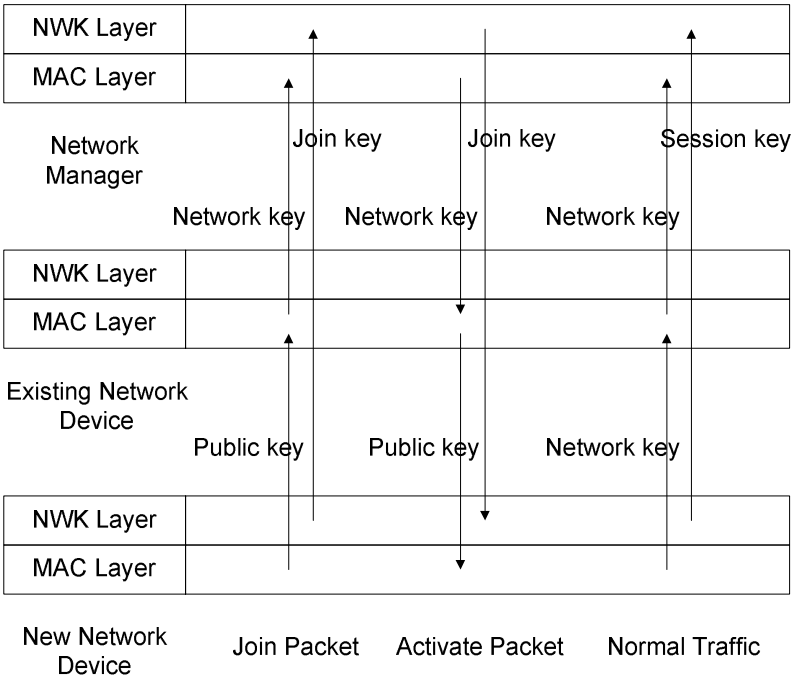


Figure 5.7: Keying Model

5.3.1 Hardware Platform

We base our implementation on the MC1321x evaluation kit [1] from Freescale. This toolkit contains one 1321x-NCB (Network Coordinator Board) board, two 1321x-SRB (Sensor Reference Board) boards, and a USB Multilink BDM Programmer/Debugger. The only major difference between 1321x-NCB and 1321x-SRB is that 1321x-NCB has a programmable 2-line LCD for displaying messages. Other than that, the two boards share the following common features:

- 40 MHz 8-bit HCS08 MCU

- 2.4 GHz wireless transceiver compatible with the IEEE 802.15.4 standard
- Programmable 60 KB Flash and 4KB RAM memory
- Multiple 16-bit timers
- USB port to interface with PC
- 3-axis acceleration sensor and temperature sensor
- 4 LEDs and switches for demonstration, monitoring and control

Together with the toolkit, Freescale also provides a simple IEEE 802.15.4 physical layer library in ANSI C. Our task is to build a new WirelessHART protocol stack by using the physical layer library.

Implicitly, this toolkit imposes two restrictions on our implementation: code size and MCU speed. The maximum code size is limited by the flash size, which is 60 KB in this case. Also, the relatively low computation capability of HCS08 makes it difficult to meet the timing requirement inside a time slot. We describe our design to solve these problems in the following subsections.

5.3.2 Timer and Timer Interrupts

WirelessHART has very stringent timing requirements on each network device. A $10ms$ time slot is further sliced into several time intervals, each of which ranges from $100\mu s$ to $4.5ms$. For example, as shown in Fig. 5.3, a receiver must start listening *TsRxOffset* time units after the beginning of

a time slot. In addition, A receiver must acknowledge a packet within *TsMaxPacket+TsTxAckDelay* time units after the arrival of the first bit of the packet. Some of the time intervals are very short. For instance, *TsCCA*, the CCA detection time, is defined to be $128\mu s$.

Also note that an ACK DLPDU is required to carry a 2-byte time adjustment field measured in microseconds. Thus, the timer used in WirelessHART MAC must be precise enough to count in microseconds.

During each time interval defined in a time slot, a node can either be idle or perform some tasks if necessary. Some of those tasks may be very time consuming. For example, when a node receives a DLPDU, it has to first verify the MIC (message integrity code) and then prepare the corresponding acknowledge. Ideally, those tasks should be finished within the designated time interval. However, in practice, the execution may take a longer time and by the time those tasks are completed, the predefined next time interval already starts. In this case, the subsequent tasks will be in serious troubles. Consequently, we cannot wait till the end of all tasks in current interval to set the next timer. Instead, we decide to make the timer aware of WirelessHART time requirements and use the timer module to start/stop the tasks.

We use a separate 16-bit TPM (Timer/Pulse Width Modulator Module) module to implement the timer. The TPM module's input clock is set to the bus clock (16MHz). By changing the internal prescaler of the TPM

module, we can change the clock frequency of the timer as follows:

$$f_{timerclock} = \frac{f_{busclock}}{prescaler}$$

Currently, the prescaler is set to 16. As a result, each tick of the timer is $1\mu s$, which is precise enough to meet the requirement of WirelessHART MAC. The TPM module contains one free running counter and one comparison counter. Whenever the free running counter equals the comparison counter, a timer interrupt is triggered.

By adjusting the comparison counter and maintaining some internal data structures, the timer module can simulate several software timers. The caller of the timer module indicates what type the next time slot would be. Then the timer module generates a sequence of timeout events in the slot based on the given time slot type (transmit/receive/idle). As an example, if current slot is a *receive* slot, the timer would first generate an interrupt at the start of the slot. Then, *TsRxOffset* time units later, it would automatically generate another interrupt to the MAC, informing the MAC to put the transceiver in the listening mode. Conceptually, an interrupt handler is composed of two parts: synchronous part and asynchronous part. The synchronous part resides in the interrupt handler, whereas the asynchronous part is included in the MAC state machine. Time critical and light-weight jobs are put in the synchronous part, and less time critical and computation intensive jobs are put in the asynchronous part. For the second interrupt in the example above, the interrupt handler only needs to set the mode of the transceiver and change

some internal states, which can all be put in the synchronous part and leave the asynchronous part empty. However, an asynchronous part is needed when some time-consuming job is incurred, such as data encryption and decryption. In this case, at the very end of the interrupt handler, a specific event is sent to the MAC state machine to signal the execution of the asynchronous part. The interrupt handler finishes immediately after that.

5.3.3 Synchronization

As time is divided into time slots and transactions within a time slot follow specific timing requirements, it is crucial that nodes in the network are kept in synchronization.

When it joins a WirelessHART network initially, a node has no idea what current time is. Fortunately, for each incoming DLPDU, a node records the time when the DLPDU's first bit arrives. Because of the strict time slot structure, a node can derive the start of the next time slot from the DLPDU arrival time according to the following formula:

$$T_{\text{next\_slot}} = \text{arrival\_time} + 10ms - \text{TsTxOffset}$$

Synchronization happens not only in the join process, but also during a node's normal operations. A receiving node always compares the start time of the incoming DLPDU and the expected arrival time measured in its own clock. The difference is the drift between their clocks. The receiver includes the difference in the time adjustment field of the corresponding ACK packet.

Each node is designated a time source node. Whenever a node receives an ACK from its time source, it will adjust its clock based on the time adjustment field. If the sender is the time source of the receiver, the receiver adjusts its clock directly from the time difference value.

5.3.4 State Machine Design

The major part of WirelessHART MAC layer is a complicated state machine. Each run of the state machine contains three steps:

1. Call the link scheduler to determine the next slot to be serviced.
2. Receive the "time slot start" event from the timer and increment the ASN by 1.
3. When it is time to service the given time slot derived in step 1), execute the associated transaction.

Most of the code in the state machine deals with executing a transaction. We define six states in the state machine:

- **Join:** In this state, the device is not authorized by the network manager yet. After successfully joining the network, it enters the Idle state.
- **Idle:** When the device successfully joins the network or finishes transmiting/receiving a packet, it enters this state.
- **Talk:** When ready to transmit a packet, the state machine enters this state and calls the XMIT engine.

- **Wait ACK:** After a non-broadcast DLPDU is transmitted successfully, the state machine reaches this state.

- **Listen:** In this state, the state machine calls the RECV engine to wait for an incoming DLPDU.

- **Answer:** In this state, the state machine constructs and sends out an ACK DLPDU corresponding to the DLPDU received in the previous Listen state.

Based on these internal states and the incoming event type, the state machine knows what task to execute. For example, when it receives a "time slot start" event, it will first increase the ASN by 1. Then, if current slot is a *transmit* slot, it sets the transceiver to transmit mode and enters into the "Talk" state.

5.3.5 Network Data Model Design

The network and transport layer maintain a set of tables, including the session table, transport table, route table and service table. These tables work together and collaborate with the tables in the MAC layer to achieve various functionalities. The interactions among these tables are summarized in Fig. 5.8.

The session table is central in the design as all the end-to-end communication in WirelessHART is built upon the concept of secure session. A

session establishes a secure data pipe between the device and one of its correspondent devices by encrypting and decrypting sent and received packets. Different types of sessions are assigned different security keys, such as session keys, join keys, and handheld keys.

The transport table is used to support end-to-end acknowledged transactions with automatic retries. In this way, WirelessHART provides reliable end-to-end in-order delivery between devices. The transport table uses a MASTER bit to identify whether the device is a MASTER or a SLAVE. Along with the corresponding sequence number, this table also buffers the payload of the last request (in MASTER mode) or response (in SLAVE mode), which allows the device to retransmit the request or response when the retry timer expires.

For each destination, there can be more than one entry in the route table with different graph IDs. When generating a network layer packet, WirelessHART has to search the route table, superframe table, and graph table to determine the routing information to be used. For certain destination, there can also be a source route, which is mainly used for network diagnostics.

Finally, the service table indicates the services associated with a certain route. All these services are allocated by the network manager and different services have different bandwidth requirements. It is the responsibility of the network manager to decide the communication schedule over the whole network to balance the traffic load.

With these well-organized communication tables and graph/source rout-

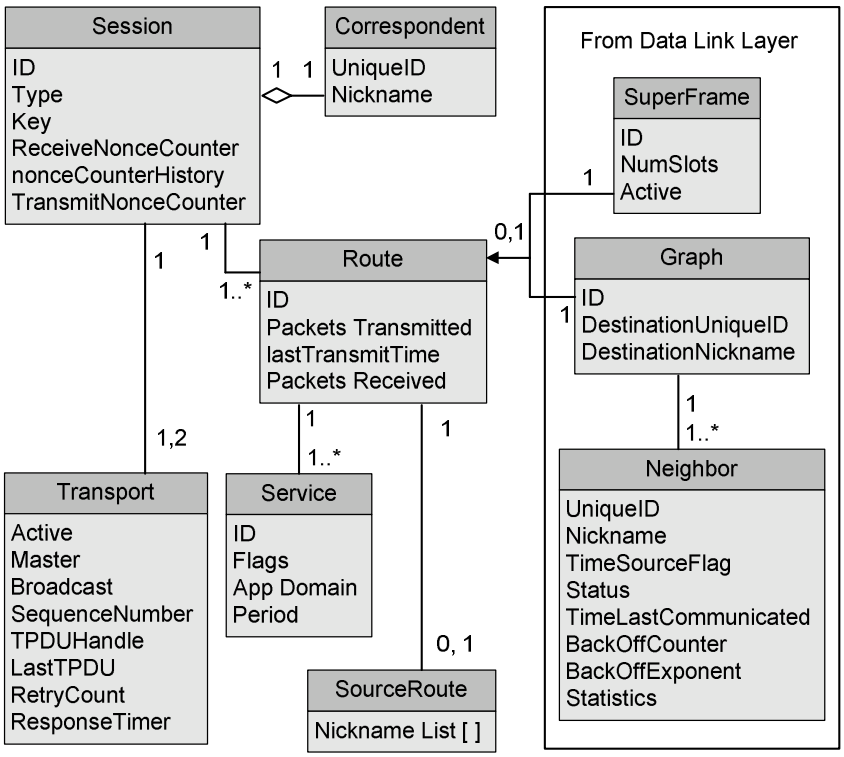


Figure 5.8: Network Layer Data Model

ing protocols, WirelessHART is able to establish secure and reliable sessions between the gateway and any device. By this means, WirelessHART supports various network topologies and provides reliable end-to-end communications.

5.3.6 MAC Layer Security

In the MAC layer, WirelessHART provides data authentication service. The authentication service uses CCM* mode (Counter with CBC-MAC (corrected)) [19] with AES-128 [6] as the underlying block cipher. CCM* needs 4

byte-strings as parameters (a, m, N, K):

- a: the additional data to be authenticated but not encrypted;
- m: the message to be encrypted;
- N: the 13-byte nonce;
- K: the 128-bit AES key.

As the DLPDU is not encrypted, the second parameter 'm' is empty, while the parameter 'a' includes the DLPDU header and payload.

The value of key K depends on the current status of the node. If a node is joining a network or broadcasting a network advertisement, the well-known key 0x7777 $772E$ 6861 7274 $636F$ $6D6D$ $2E6F$ 7267 is used. In all other situations, the network key assigned by the network manager is used.

The nonce N is the concatenation of the absolute slot number and the source address. The first 5 bytes are always the absolute slot number. If the source address of the DLPDU is a long address (EUI-64 address), the source address is filled into the remaining 8 bytes of the nonce. Otherwise, the short source address (2 bytes) is put right next to the slot number, with the rest 6 bytes filled with 0.

The sender and receiver of the DLPDU both call the *CCM** function with the same input: the DLPDU header and payload. After receiving a DLPDU, the receiver compares the returned MIC with the MIC in the original

message. If they match, the message is authenticated. Otherwise the message is invalid and discarded.

From the perspective of a receiver, it must run *CCM** on the received message and on the corresponding ACK message within $TsTxAckDelay(1ms)$. This is a very challenging task for low power processors.

We grabbed a complete *CCM** package from [5]. This program only encrypts/decrypts one block of data, which is 16 bytes. There are three major underlying functions in the program: *aes_set_key*(), *aes_encrypt*(), and *aes_decrypt*(). We measured the execution time of these three functions on the hardware platform described in previous section. On average,f *aes_set_key*() takes $1341\mu s$, *aes_encrypt*() takes $1335\mu s$, and *aes_decrypt*() takes $1581\mu s$. The sum of these numbers exceeds the timing limit of WirelessHART. It should also be noted that those functions only works on 16 bytes, while a WirelessHART DLPDU can be as long as 121 bytes.

To meet the stringent timing requirements of WirelessHART, we acquired a more powerful toolkit from Freescale: DEMOJM128 kit. The run time for the three major functions *ccm_init_and_key*(), *ccm_encrypt_message*(), *ccm_decrypt_message*() are $77\mu s$, $795\mu s$, $818\mu s$, respectively. Obviously, without any tweak, the DEMOJM128 toolkit would not meet the requirement of WirelessHART either.

We propose to use a hardware accelerator to speed up the encryption/decryption process. Upon our request, Freescale is developing a new

toolkit with on-board hardware accelerator. Freescale claims that the encoding of 16 bytes of data would be finished in 13 system clocks with the new toolkit. However, even with the new toolkit, we may still have problems. Except for the $aes\_encrypt()$ function, $ccm\_encrypt\_message()$ and $ccm\_decrypt\_message()$ also call some other helper functions. Of the $795\mu s$ and $818\mu s$ run times of the two functions, the helper functions take about $123\mu s$ and $148\mu s$, respectively. These times cannot be improved by the hardware accelerator.

In order to further speed up the encryption/decryption process, we propose to execute *CCM** incrementally. Originally, *CCM** is not designed to support stream processing. However, as WirelessHART DLPDU is not encrypted, the message length is indicated in the DLPDU header. Thus, we can run *CCM** on an incoming message as soon as every 16 bytes are received. Given the relatively slow data transmission rate (250kbps), we may only need to process one block of data in the $TsTXAckDelay$ period, regardless of the message length. In this way, we can meet the stringent timing requirements of WirelessHART.

5.3.7 Network Layer Encryption and Authentication

WirelessHART also provides built-in security support in the network layer. A keyed MIC is used to ensure that the NPDU arrives successfully and unmolested from the indicated source device. Similar to that in MAC layer, the MIC is generated and checked using *CCM** mode in conjunction with the AES-128 block cipher. For each session, four critical fields are kept

up-to-date for NPDU encryption and authentication: *sessionKey* records the 128-bit write-only session key; *peerNonceCounter* maintains the largest nonce counter value received from the correspondent device; *myNonceCounter* is the nonce counter for packets sourced by the device, and *nonceCounterHistory* is an array of bits recording the nonce counters received. The most significant bit of *nonceCounterHistory* is always set and corresponds to the current *peerNonceCounter* value.

In the process of enciphering the NPDU, the NPDU payload to be enciphered is the byte-string 'm'. The NPDU header is the byte-string 'a'. In the byte-string 'a', the TTL, Counter, and MIC fields are set to zero and will be replaced with their actual values before transmitting the packet. The corresponding 128-bit session key is the byte-string 'K' and the 13-byte network layer nonce 'N' is constructed differently depending on the transmission types.

If a packet is a join response, $N[0]$ is set as 1 and we use the *peerNonceCounter* value (from the join request) as the nonce counter and load the joining device's EUI-64 address into the nonce. In other cases, we set $N[0]$ as 0. *myNonceCounter* in the session would be pre-incremented by one and written to the nonce. Then the NPDU source address field (EUI-64 address or nickName) is loaded into the nonce.

To authenticate a received packet, at the destination device, the NPDU nonce is re-constructed and the packet is deciphered. If the NPDU is a join response, $N[0]$ is set to 1, otherwise it is set to 0. The NPDU source address field is loaded into the nonce. If the message is a join request, the NPDU

counter is four-bytes long and would be copied to the nonce counter. On the other hand, if it is a join response, the NPDU counter would be compared to *myNonceCounter*. If they do not match, the packet is discarded. Otherwise, the *myNonceCounter* is copied to the nonce counter. Under all other cases, the NPDU counter is one-byte and the nonce counter is re-constructed. To do this, the most-significant three bytes of the *peerNonceCounter* are copied to $N[1] - N[3]$. If the NPDU Counter value is less than the quantity (1+LSB(peerNonceCounter) - sizeof(nonceCounterHistory)) then the 24-bit value in $N[1] - N[3]$ is incremented. The NPDU Counter is copied to $N[4]$.

The resulting nonce counter is compared to the nonceCounterHistory. If it corresponds to any of the bits set in the nonceCounterHistory, the packet must be discarded. Otherwise, the nonceCounterHistory will be updated accordingly. This finishes the packet authentication process.

5.3.8 Command Processor in the Application Layer

Figure 5.9 describes our architecture of the application layer. Since the application layer is command-oriented, the core module is a command processor. When a field device receives an incoming command request, the application layer will first parse the data message from the network layer into a command message (APDU) and relay it to the command processor. Based on the command number in the APDU, the command processor will choose the correct command handler to generate corresponding response. The command handler communicates with other layers through an interwine service

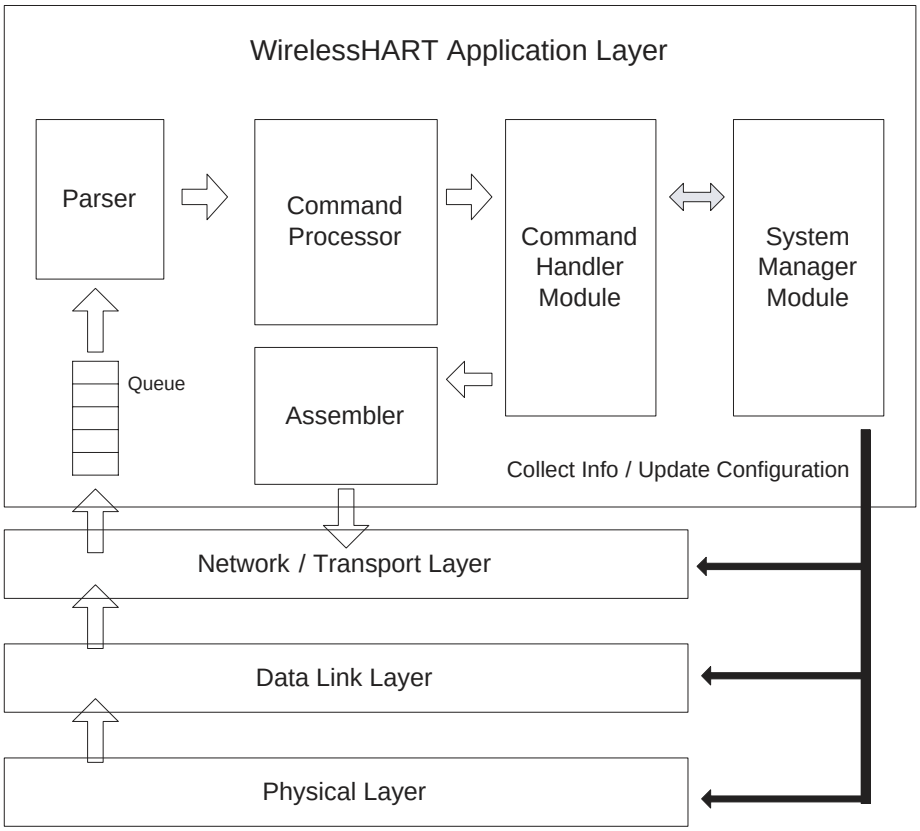


Figure 5.9: WirelessHART Application Layer Architecture

module to collect necessary information (for example, the number of links in the MAC layer) or update configurations (e.g., write a route to the network layer). Finally, the command handler assembles these information and generates the command response to the application encapsulator, which in turn sends back to the network layer.

5.3.9 The Join Process

Before a new node can be integrated into a WirelessHART network, it must go through the join process. Through the join process, a node is provisioned with a short address (nickname), a network key for MAC layer authentication, a session key for network layer security, and most important, some links and routes to allow the bi-directional communications with the network manager.

To be able to start the join process, a new node should be first programmed with a network ID and a join key. The network ID determines which network the device will join, and the join key is a password presented to the network management to be admitted to the network.

When the device is instructed to start the join process, the MAC layer is placed in a continuous listening mode. By capturing a network advertisement with the pre-programmed network ID, the node gets to know the start of a time slot and the network wide absolute slot number. It can keep polishing its measurement of the start of a time slot with subsequently received advertisements. After the MAC layer fixes the timing, it starts to forwards captured advertisements, together with the received signal strength, to the network layer. After gathering enough advertisements, the network layer decides which neighbor to send join requests to, usually based on the received signal strength and join priority associated with each advertisement.

Then the MAC layer assembles a join request and sends it through the

join link as indicated in the advertisement from the selected neighbor. The selected neighbor then forwards the join request to the network manager. The network manager checks the unique address and join key in the join request. If it deems the join request is valid, the network manager can grant the request by allocating a short address and sending the network key to the joining device. The new device shall use the network key thereafter. In addition, the network manager can write some superframes and associated links to the device, so that the new device can start to publish its sensing data periodically. This concludes the join process of a new node.

The critical step in the join process is time synchronization. As the advertisements are broadcasted using the channel hopping mechanism, the channels on which the new device listens should also change frequently. In our implementation, the device scans through all channels sequentially and stays on each channel for 400 ms. Another complication is the handling of channel blacklists. Special care must be taken if an advertisement contains a channel blacklist. In this case, the channels in the list should be skipped.

It is also important to switch from the join key to the network key and from the join links to normal links at correct time. It turns out there is a simple rule: always use the most updated resources. That is, a joining device keeps using the join key, until it receives a network key from the network manager; the device keeps sending/receiving packets on join links till it is configured with normal links.

5.3.10 Network Management

According to WirelessHART, two most important functions of a network manager are generating routes and communication schedules. Although WirelessHART specifies what a network manager should do in various cases, it leaves out all details. That is, the network manager has the freedom to choose how to fulfill its tasks. For example, when generating routes, the network manager can aim to either balance the load on network nodes or minimize the average network latency. Depending on the metrics the manager tries to optimize, the resulting routes can be very different.

For the purpose of proof of concept, we implemented a simple network manager. We illustrate the design of the network manager by an example. The topology of the example network in Figure 5.4 is shown in Figure 5.10, where the number below each node is the node's identification number. The burst mode data rate from each device is summarized as follows:

Device ID	3	4	5	6	7	8
Burst Mode Rate (seconds)	8	32	4	64	16	64

5.3.10.1 Generating Routes

Based on the link information provided by each node, the network manager needs to create the overall network graph. In this process, the manager follows several rules in the specified order:

(a) Minimize the number of hops.

(b) Route through powered devices if they are available.

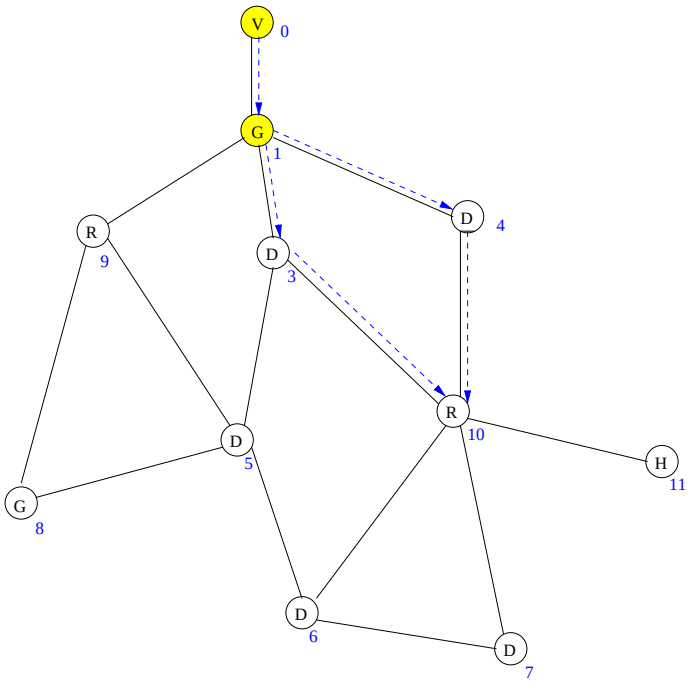


Figure 5.10: Topology of the WirelessHART Network in Fig. 5.4

(c) Use signal strength to select the best paths to neighbors.

(d) Use a combination of weighted signal strengths to select between alternative routes.

(e) Prune the number of neighbors to 4 or less.

After generating the overall network graph, the manager has to create the follow graphs:

- An graph describing paths from each network device to the gateway
- A broadcast graph from the gateway downward to each device

- Graphs from the gateway to each device

5.3.10.2 Generating Communication Schedules

The network manager has to assign time slots to each device according to the graphs derived above. The strategies we take are summarized below:

(a) The network management superframe has priority over data superframes.

(b) Traverse the graph by breath-first search, starting from the gateway, number the devices as $N_0, N_1, \ldots, N_n$.

(c) Every device needs to have a slot for a keep-alive message. The keep alive timer is 60 seconds.

(d) For join requests, from the furthest devices, allocate one link for each en-route network device to the gateway (No redundancy provided)

(e) For join responses, traverse the graph by breath-first search, allocating one link for each en-route network device from the gateway to end network device.

(f) For data requests, the allocation is similar to join requests. An additional slot is allocated in each en-route device for retransmission.

(g) If there is an alternative path, allocate a slot for it.

The resulting schedule is shown in Figure. 5.11. The upper part of the figure shows the overall slot allocations divided into logical channels. The

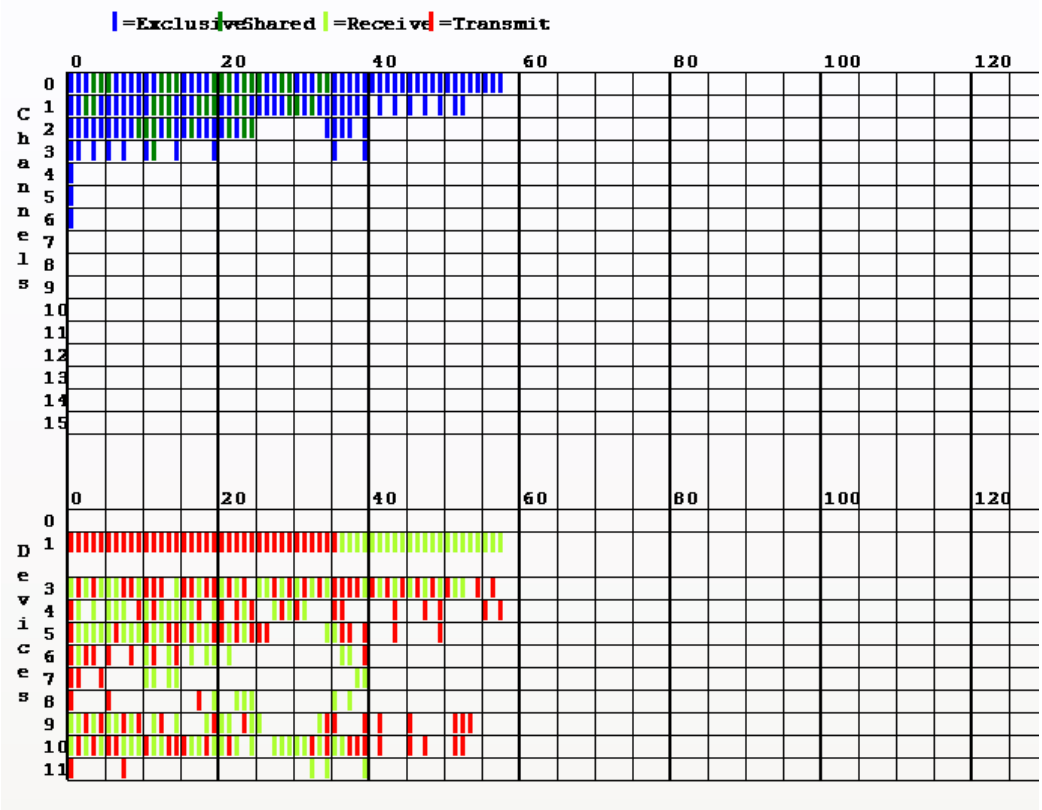


Figure 5.11: The Overall Schedule for the Sample Network

bottom portion of the schedule describes the transmit slots and receive slots for each device.

After the overall schedule is generated, the network manager splits the schedule into sub-schedule for each device and distributes them to the corresponding devices.

5.4 A WirelessHART Network Demonstration

Based on the WirelessHART prototype stack, we build a demonstration network [49] with the Freescale DEMOJM128 toolkit. The application is written in ANSI C.

Compared to the MC1321x evaluation kit, the DEMOJM128 kit is more powerful. It has a *ColdFire V1* 32-bit core (up to 50.33MHz), 16KB RAM and 96KB flash. As usual, it also includes BDM support for in-circuit debugging.

The demonstration network include one central network manager, a gateway, one actuator and two sensors. The actuator and sensors are shown in Fig. 5.12. The network manager is based on Wi-HTest[26] and talks to the actuator/sensors through the gateway. All devices first join the network through the gateway. After joining the network, *Sensor 0* sends its data to the gateway every 2 seconds. *Sensor 1* is instructed by the network manager to send its sensing data first to the actuator, which would then forward the packets to the gateway. These operations can be easily implemented by the manager to write special routes and graphs to *Sensor 1* and the actuator. Sensor 1 generates a packet every 3 seconds. On the other hand, the gateway sends commands to the actuator every 20 seconds.

Every time a sensor transmits a packet, it blinks its LED. For the actuator, it flashes its LED when it receives a packet. In addition, to examine the exact interactions among these devices, we use WiAnalysis [26] to capture the packets flying in the air.

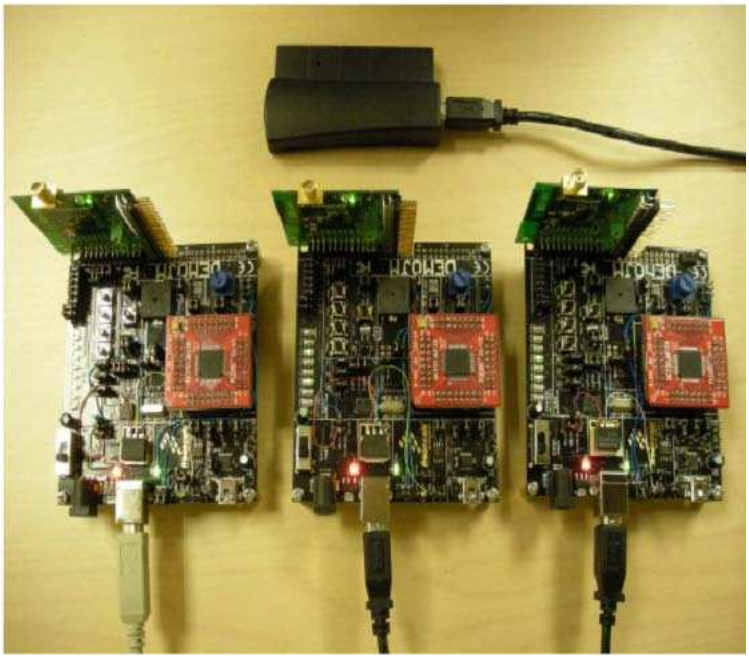


Figure 5.12: The Demonstration Network

5.4.1 The join process

In our experiments, each device can join the network without any problem. As an example, the join process of *Sensor 0* is shown in Fig.5.13. For clarity, we only show those messages related to the join process.

After powering up, it first has its clock synchronized to the network by decoding advertisements from the gateway. Then, it sends a join request (Packet 732) to the gateway. Note in this case, the device is using its unique (long) address. As a response, the gateway sends Packet 751, which informs the new device its nickname (short address) and the network key. In subsequent packets, the device starts to use the nickname and network key. Then, the

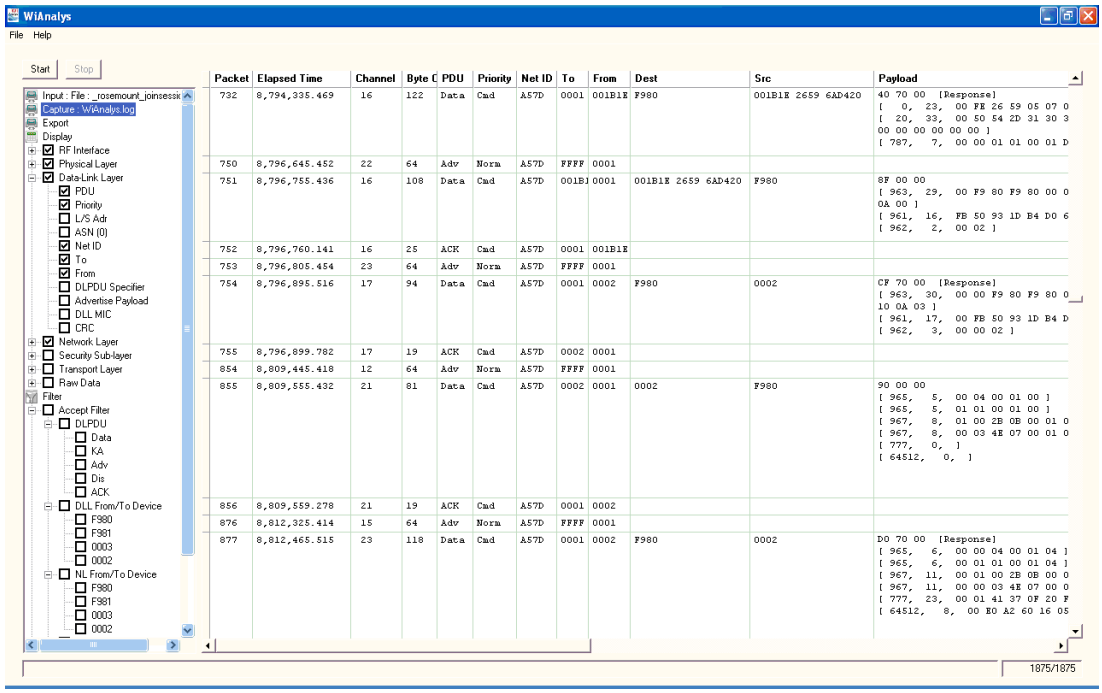


Figure 5.13: Sensor 0 joins the network

network manager starts to write some superframes and associated links (Packet 855) to the device. After the device sends out the response packet 877 to confirm accepting the superframes and links, the device is considered to be integrated to the network.

5.4.2 Sensing and Actuating

For a device to be fully functional, it must be configured with routes and graphs. Basically, the device needs to know how to handle a packet if the destination address of a packet is not the device itself.

In this demonstration, we program the network manager carefully to

configure the network in such a way that *Sensor 1* sends its outgoing packets to the actuator, which in turn forwards them to the gateway. On the other direction, the actuator routes the packets from the gateway to *Sensor 1*.

We capture all the packets exchanged between these devices and find they act exactly as described above.

In order to gather detailed performance parameters for a WirelessHART network, a powerful network manager is required, which is our focus in the near future.

5.5 Discussions

WirelessHART is the first open wireless standard for industrial process control. To meet the strict real-time requirements of WirelessHART, we proposed some novel ways to tackle the challenges (for example, timer design, synchronization and security, etc.) and built a demonstration network to verify our design.

The network manager is the central control unit of a WirelessHART network. When deriving the routing table and communication schedules, the network manager can choose to optimize several metrics, such as energy consumption, average end-to-end latency. We believe the scheduling algorithms can be vastly different depending on the optimization goal. For now, our network manager is a proof of concept and far from perfect.

Chapter 6

Adaptive PID algorithm in Wireless Process Control

In previous chapters, we treat a process control system as a multiprocessor system with communications. Basically, we model a function block as a periodical task with release time, execution time and deadline requirements. In this way we can simplify the problem and treat it as a real-time multiprocessor scheduling problem. We are essentially mapping the original problem to a real-time multiprocess scheduling problem. As we pointed out before, wireless communication is susceptible to interferences and noises and is inherently unreliable. In case of excessive communication failures, our assumption about the system does not hold any more and we suggested resorting to other emergence mechanisms, for example, the function block entering a fail-safe mode.

In this chapter, we try to tackle this unreliable communication problem from the perspective of control systems. We look into the detail of control blocks and propose to enhance the control blocks to better handle lost communications.

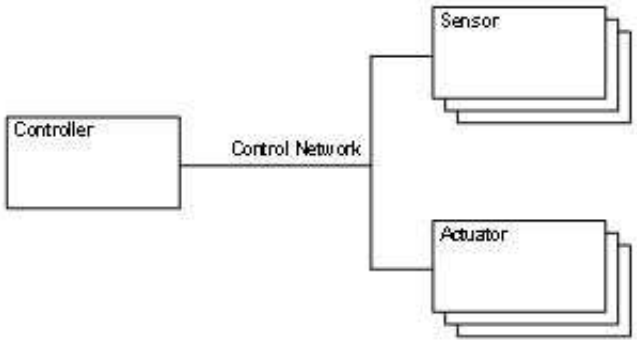


Figure 6.1: A process control system

6.1 Introduction

A modern process control system is often structured as shown in Figure 6.1. As illustrated in this figure, sensors and actuators are connected to controllers via control networks, such as Fieldbus [23] and Profibus [44]. A sensor provides measurements and status of a physical property, e.g. the flow in a pipe associated with the process. Based on the measurement from sensors, the controller determines any adjustment in the actuators that is needed to maintain the process at a target value, i.e., the setpoint. The control loop is executed periodically at a rate fast enough to correct any unwanted deviations in the process.

In contrast to non-wireless control systems, communications in a wireless control system are inherently unreliable. This unreliability may be caused by interferences, power failures and environmental factors such as lightening storms. Unfortunately, current control strategies are based on the assumption

of reliable communications. In standard process control systems, a missed IO communication is considered an error. Usually, a control loop is configured with a maximum number of lost IO communications, after which the loop declares failure and the values are set according to the fail-safe configuration.

Therefore, unreliable communications presents a very challenging problem for standard control paradigms. Consider a PID (Proportional, Integral, Derivative) block with inputs from a wireless channel. Suppose the inputs are lost at time $t1$ and reestablished at time $t2$. The derivative component of the PID would cause a spike in the output at $t2$. Also, from $t1$ to $t2$, the reset component may base on the error that existed at time *t1*.

In this chapter, we focus on the most widely used control blocks - PID blocks. We improve the calculation of the integral and derivative components of PID algorithms by detecting missed communications and compensate for it proactively. Simulation results on several wireless scenarios prove the superiority of the enhanced PID algorithm.

The rest of this chapter is organized as follows. Section 2 analyzes the shortcomings of the standard PID block in dealing with intermittent communication losses. Section 3 describes the enhanced PID algorithm. Experiments and results are presented in Section 4. Section 5 concludes this chapter.

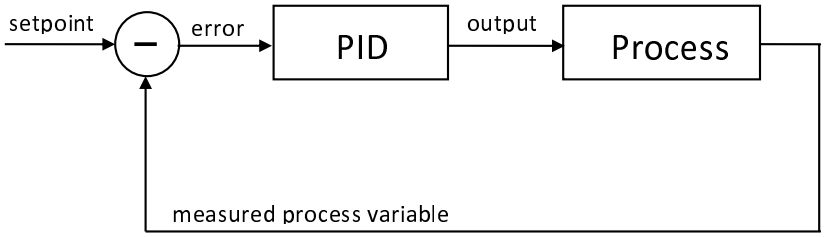


Figure 6.2: PID block

6.2 The standard PID algorithm

PID is the most widely used control algorithm in industrial process control [33]. As shown in Figure 6.2, the controller compares the process variable (PV) with a reference setpoint (SP). The error is then processed to calculate a new output to bring the PV back to its desired SP [43].

PID stands for "proportional, integral and derivative" components of the algorithm. Each of the three components performs a different task and has a different effect on the functioning of a system. Their outputs are summed up to produce the system output.

Though there are many variations of PID algorithms, in its non-interacting form without rate limiting and all actions based on error, the equation for standard PID algorithm is

$$Output = K_P \left[e(t) + K_I \int e(t)dt + K_D \frac{de(t)}{dt} \right] \tag{6.1}$$

K_P, K_I, K_D are the proportional, integral and derivative gains, respectively.

In its digital form, the software implementation of the PID algorithm

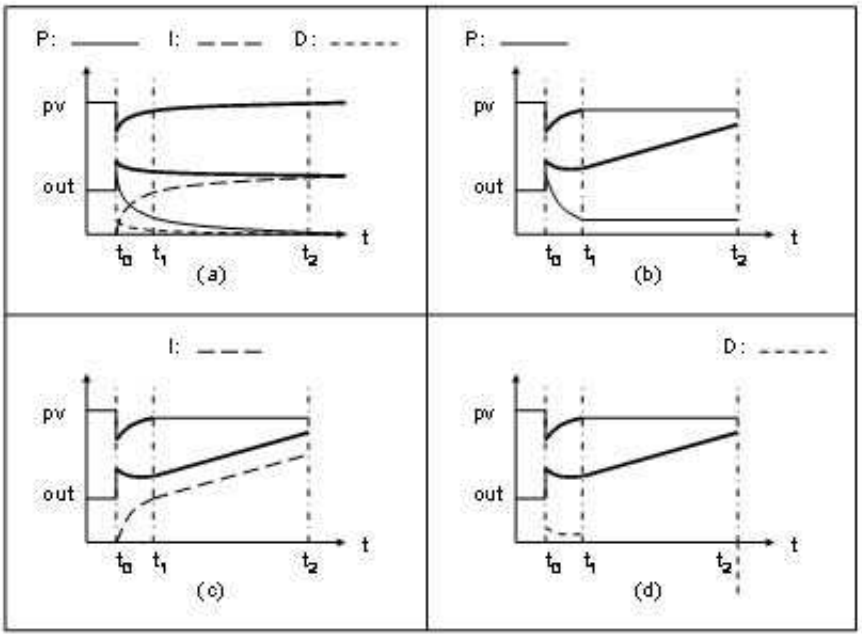


Figure 6.3: Standard PID block with lost input

is based on the sampled data for its PV on a periodic basis.

When there is no communication loss, PID, once configured for the process it controls, keeps the process in a steady state. Figure 6.3(a) shows the PID reaction to a process disturbance.

Before time $t0$ the PID output (out) is kept at a constant value to maintain PV at the SP value. At $t0$ a drop of PV is observed due to process disturbances. To correct the drop, the PID increases its output. At time $t2$, PV comes back at SP, and out is stabilized at some value that is slightly bigger than its original value to compensate the disturbance. As is shown in

Figure 6.3(a), *out* is the sum of three parts: P, I, and D.

6.2.1 Input Communication Lost

Now consider what would happen to each component of *out* if the communication from the sensor input is lost between times $t1$ and $t2$. For the PID block, the measured PV value remains the same as that at time $t1$ during this period, shown in Figure 6.3(b, c, d).

Figure 6.3(b) shows the proportional gain P. Since the measured PV and SP remains the same, the proportional gain is constant from $t1$ to $t2$. Figure 6.3(c) shows the integral part I. Since PV and SP remains the same, the error remains the same, so the integral part is a linear increasing line from $t1$ to $t2$. Figure 6.3(d) shows the derivative part D. Since PV and SP remains the same, the error remains the same, so the derivative stays 0 from $t1$ to $t2$. As a result, *out* of the PID block is a linear increasing line from $t1$ to $t2$, shown in Figure 6.3(b, c, d). This destabilizes the process. The longer the communication is lost, the bigger deviation PV is from SP.

Once the communication is reestablished at $t2$, PID is back to normal. However, since the derivative part calculated at time $t2$ is based on the difference of measured PV's between $t2$ and one period before $t2$, we shall expect a spike for the derivative part. This is because the PV value at $t1$ is used as the PV value at one period before $t2$. The value of PV at $t2$ could be significantly different from the PV value at $t1$. Since PV changes between time $t1$ and $t2$, we expect the derivative spike even bigger. Due to sudden changes of

the proportional and derivative parts, the value of *out* will have a big impulse before and after $t2$.

6.2.2 Output Communication Lost

We continue to analyze how standard PID behaves if output communication is lost between $t1$ and $t2$. Here we assume there is no other disturbance and input communication is stable.

From time $t1$ the actuator will stay with the *out* value of $t1$ until $t2$ when a new *out* value is received from the PID. This will cause the measured PV to eventually reach SP and then overshoot a little bit, shown in Figure 6.4(b, c, d). The P, I, and D components are all calculated based on the current PV, shown in Figure 6.4(b, c, d). This result is as good as we could expect from PID. The only drawback is that the actuator gets a bump in the *out* value, from the one calculated at $t1$ to the one calculated at $t2$.

6.2.3 Both Input and Output Communication Lost

When both input and output communications are lost, the PID behaves the same as when only input communication is lost in Figure 6.3. The only difference is that when communication is reestablished at $t2$, the actual PV is different. In this case PV strays less as the actuator output stays constant. Similarly, the actuator receives a bump in the out value.

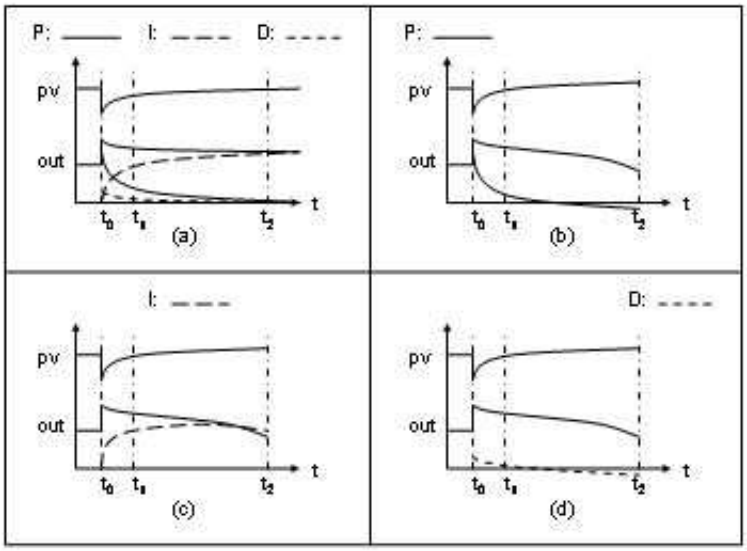


Figure 6.4: Standard PID block with lost output

6.3 The PIDPlus Algorithm

The underlying assumption in the digital implementation of the PID algorithm above is that the algorithm is executed on a periodic basis. When the input containing the measurement is lost, the calculated reset action may not be appropriate [15]. When later on a new measurement gets through, the calculated derivative action may produce a spike in the output. If a PID block continues to execute using the last process variable, the output will continue to move based on the reset tuning and error between the last measured process variable and the setpoint. If the control block is only executed when a new measurement is communicated, then this could delay control response to setpoint changes and feedforward action on measured disturbances. Also, when

control is executed, calculating the reset contribution based on the scheduled period of execution or on the time since the last contribution may result in changes that increase process variability [15].

In [15], we proposed an enhanced PI algorithm to reduce wireless communications between sensors and controllers without impacting the control performance significantly. Based on that work, we further improve the derivative part and apply the new algorithm to address communication losses.

To provide best control when measurements are not updated on a periodic basis, the PID may be restructured to reflect the reset and derivative contributions for the expected process response since the last measurement update. One means of doing this is illustrated in Figure 6.5. As shown in Figure 6.5, the reset/rate contributions (integral/derivative parts of the PID) are determined based on the use of a new value flag from the communications stack, the same idea as in [15]. To account for the process response, the filter output is calculated in the following manner when a new measurement is received:

$$F_N = F_{N-1} + (O_{N-1} - F_{N-1}) * \left(1 - e^{\frac{-\Delta T}{T_{reset}}}\right) \tag{6.2}$$

where

F_N = New filter output,
F_{N-1} = Filter output for last execution,
O_{N-1} = Controller output for last execution,
ΔT = Elapsed time since a new communication was communicated.

Since the last communicated actuator position as reflected in the feed-

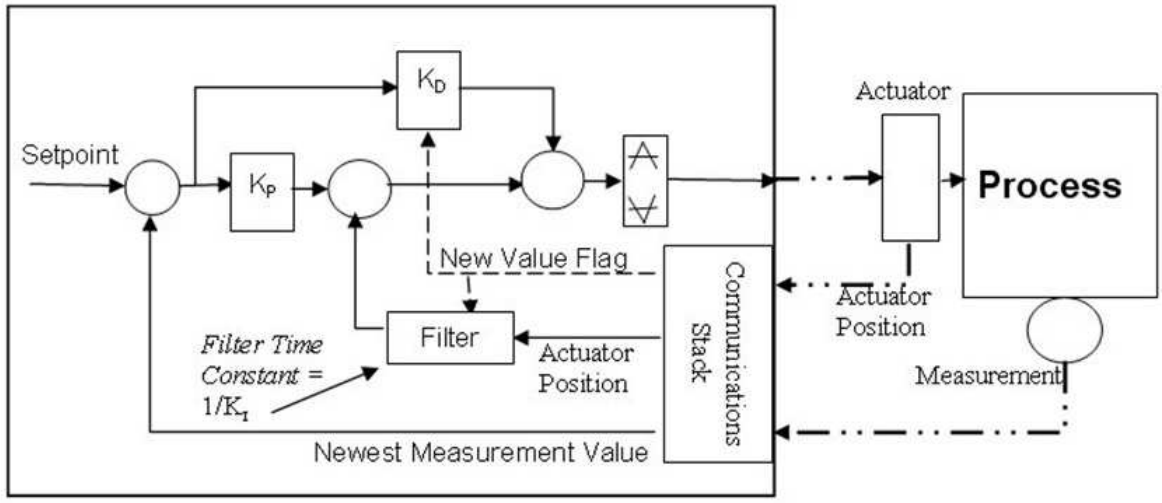


Figure 6.5: The enhanced PID algorithm application

back of actuator position is used in the integral calculation, this automatically compensates for any loss in the output communicated to the downstream element.

The derivative part for this example (rate limiting not applied) is determined by the following equation

$$O_D = K_D \times \frac{e_N - e_{N-1}}{\Delta T}$$

where e_N = current error, e_{N-1} = last error, ΔT = Elapsed time since a new value was communicated, O_D = controller derivative term.

Consider the contribution of the derivative part when the inputs are lost for several periods. When the communication is reestablished, $e_N - e_{N-1}$

in the equation above would be the same for the original and modified algorithms. However, for the standard PID algorithm, the divisor in the derivative part would be the period, while that in the new algorithm is the elapsed time between two successfully received measurements. It is obvious that the modified algorithm would produce smaller derivative action than the standard PID algorithm.

There are two major problems with standard PID algorithm when dealing with communication losses: continued execution during communication loss, and sudden output change when communication is reestablished. The enhanced PID algorithm solves these problems by only computing the integral and derivative components when communication is established and incorporating actuator feedback into the reset calculation.

6.4 Experiments and Results

We carried out several experiments to validate our algorithm. First, we prove that the new algorithm will produce the same result as the regular algorithm when the communication is reliable. Then, the revised algorithm is shown to have better performance than the existing PID algorithm when communications are unreliable.

6.4.1 Experimental Setup

We create two simple PID control loops, as shown in Figure 6.6.

$PROC_1$ and $PROC_2$ are two identical processes, each of which consists

of a second order process with a delay of 1 second and time constants of 6 seconds and 3 seconds.

The modified PID algorithm is implemented in PIDPLUS, while PID2 is a standard PID block. The parameters for PID2 are determined by testing it with a tuning application, which suggests a gain of 0.85, a reset of 10.71 and a rate of 1.71. Then the tuning parameters of PIDPLUS are set the same as PID2. PIDPLUS is configured to utilize the $BKCAL_IN$ value for the reset components.

The process variable communication is simulated by the *COM_IN_1* and *COM_IN_2* block, which are controlled by *COM_STATUS_IN*. If *COM_STATUS_IN* is set to 1, block *COM_IN_1* and *COM_IN_2* relay measurements accurately. Otherwise, the two blocks drop the measurements. The same logic is applied to the output using *COM_OUT_1*, *COM_OUT_2* and *COM_STATUS_OUT*.

By changing the external setpoint and introducing some disturbances that impact each PID and associated process equally, we can evaluate the performance of the two PID blocks. The performance of the two PID blocks is collected in the $PERFORMANCE$ block. The metrics used in this chapter is Integral Absolute Error (IAE).

The scan rate for all blocks is set to 0.2 second. Initially, the uncontrolled disturbance (DISTURBANCE) to the processes is 20.

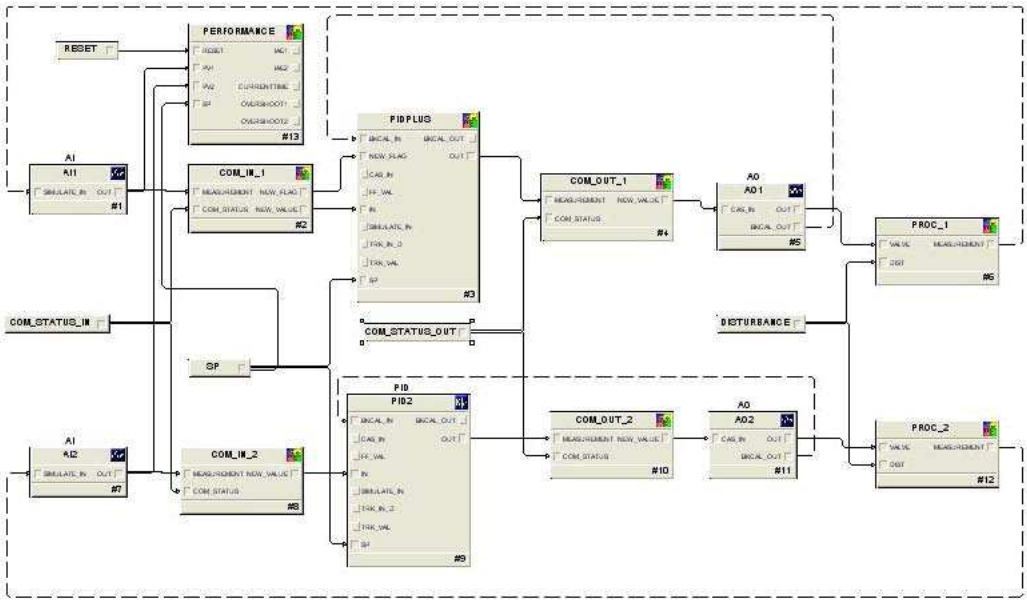


Figure 6.6: Experimental Setup

6.4.2 Reliable Communication

To simulate the case of reliable communications, we set both *COM_STATUS_IN* and *COM_STATUS_OUT* to 1. *SP* is changed from 50 to 60. The result is shown in the left part of Figure 6.7 (from time 11 : 10 to 11 : 11).

The curve of AI1/OUT.CV matches nicely with that of AI2/OUT.CV, and AO1/SP.CV matches nicely with AO2/SP.CV. Therefore, we conclude that the two PID blocks work in the same way when the communication is reliable.

6.4.3 Unreliable Communication

To study the effect of unreliable communications on the two PID blocks, we consider two cases: unreliable input and unreliable output.

6.4.3.1 Unreliable Input

During the period of lost inputs, the last communicated process variable is maintained and used in the PID blocks. We first experiment with changing setpoints. The result is shown in the right part of Figure 6.7, where SP is decreased from 60 to 50.

When the input channel is shut down, the error between the setpoint and process variable fed to PID blocks is a constant. For the standard PID block, PID2, the integral part would keep integrating, which explains the linear decreasing of *AI2/OUT* and *AO2/SP*. However, as it has a flag for missed communications, PIDPLUS would simply freeze the reset component during loss of communication, which explains the level-off in AO1/SP during lost communications. Since *AO1* is given a constant value, the process variable of *PROC_1* approaches the setpoint gradually, as shown by *AI1/OUT*.

When communications are re-established, the input process variables to *PIDPLUS/PID2* reflect the true measurement provided by *AI1/AI2*, respectively. For PID2, *AI2/OUT* is very low compared to the setpoint, which causes a sharp spike in *AO2* by the derivative part of PID2 at the moment communications are re-established. For PIDPLUS, *AI1/OUT* is close to the setpoint, and that small deviation is further evened out by the divisor used in

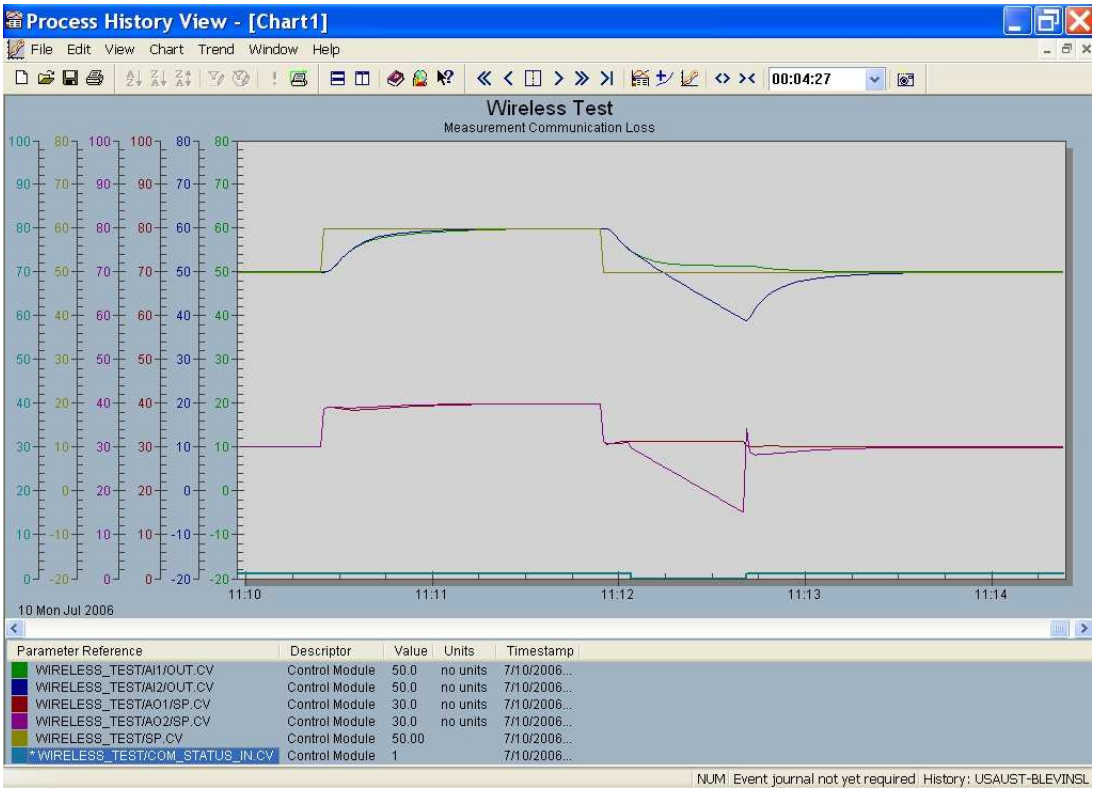


Figure 6.7: Lost Inputs coupled with a setpoint change

the derivative part of the new algorithm. Thus both *AI1* and *AO1* transit to their steady states smoothly.

The different behaviors of the two PID blocks are further proved by the performance data. In duration of 121 seconds, the IAE for PIDPLUS is 169, while that for PID2 is 372.

We also test the two PID blocks with unmeasured disturbances. The DISTURBANCE is increased from 20 to 30. The result curves are shown in Figure 6.8. The behavior of PID2 is the same as in Figure 6.7, which can

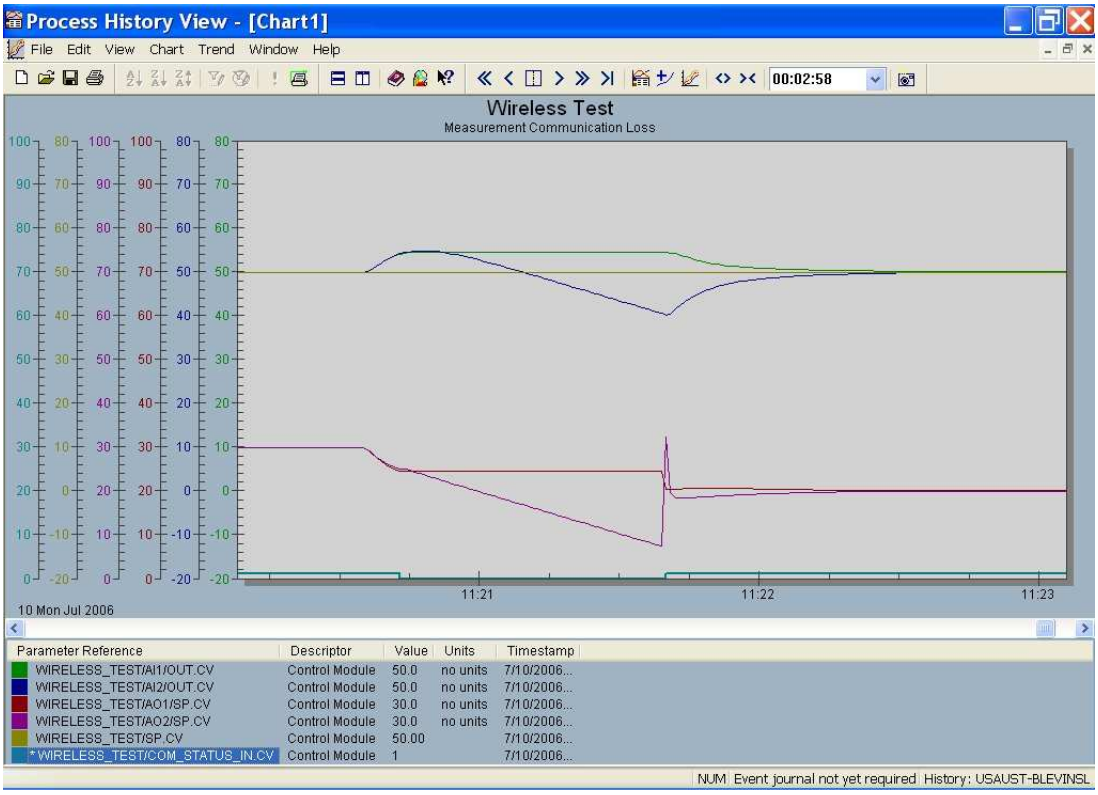


Figure 6.8: Lost Inputs coupled with unmeasured disturbances

be explained by the same reason for Figure 6.7. For PIDPLUS, the reset component is maintained constant during the loss of communications. Thus AO1 is kept the same output value, which in turn produces the same AI1 value. When the input communications resume, PIDPLUS starts to change its output (*AO1/SP*) to bring *AI1/OUT* back to the setpoint. During the time (196 seconds), the IAE for PIDPLUS is 333, and that for PID2 is 366.

6.4.3.2 Unreliable Output

In this case, we examine the behaviors of the two PID blocks in two scenarios too: setpoint changes and uncontrolled disturbances.

For setpoint changes, we first change the setpoint from 50 to 60, when the communication is reliable. Then, after the processes settle at the setpoint 60, the setpoint is changed back to 50, and the output channels are cut off. Figure 6.9 shows the resulting curves.

When the communication is lost, the outputs of *PIDPLUS/PID2* are equal. Since the two controlled processes are the same, the values of *AI1.OUT* and *AI2.OUT* follow the same curve. When the communication is reestablished, the input errors for PIDPLUS and PID2 are the same. However, the divisor in the derivative part of PIDPLUS is much bigger than that in PID2, which explains the sharper spike in the curve for *AO2/OUT*. During the transition period, the IAE for PIDPLUS is 190, while that for PID2 is 196.

Again, we test the two blocks by introducing uncontrolled disturbance to the processes. The DISTURBANCE is changed from 20 to 30. The results are shown in Figure 6.10.

The curves in Figure 6.10 is similar to their counterparts in Figure 6.9 and they can be explained the same way as above. We also notice that the improvement of PIDPLUS over PID2 is more pronounced in Figure 6.10 than in Figure 6.9. This is because at the time of communication reestablishment, the input errors to the PID blocks are bigger in Figure 6.10 than in Figure 6.9.

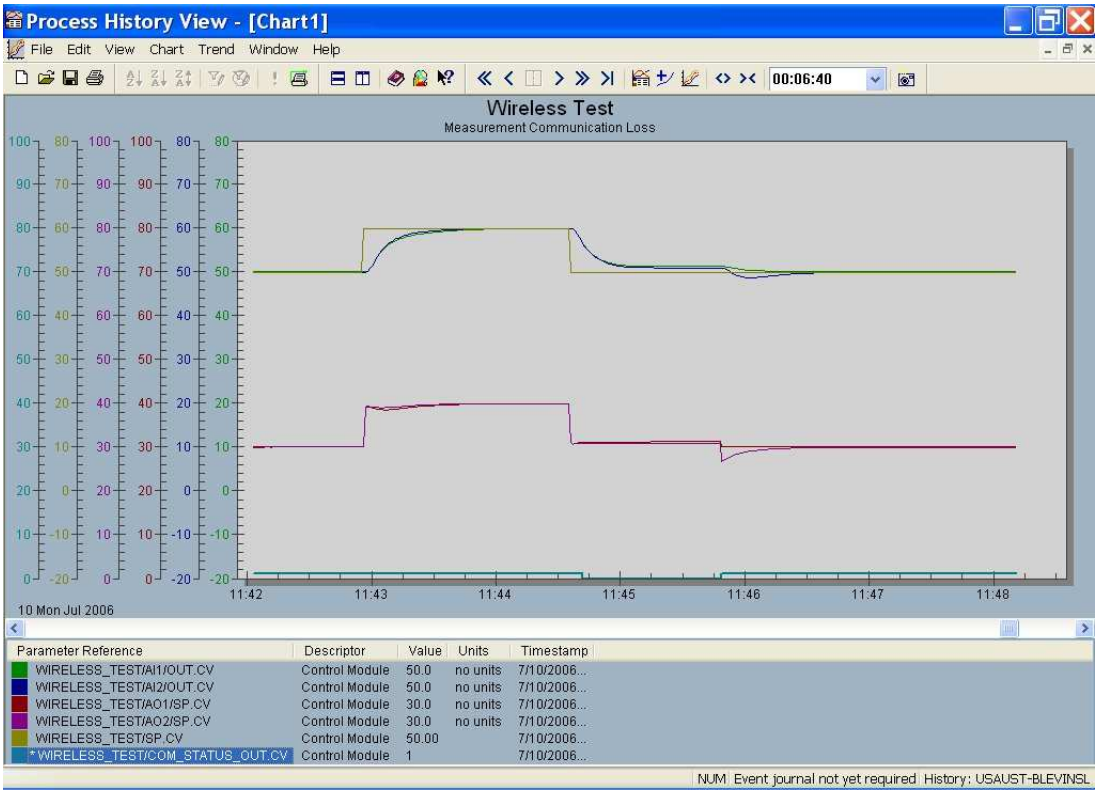


Figure 6.9: Missed Outputs with a setpoint change

During the transition period (158 seconds) in Figure 6.10, the IAE for PIDPLUS is 267, while that for PID2 is 388.

6.5 Conclusions

Using actuator feedback for smooth output transition is not a new idea. The Foundation standard already defines back-calculate-in/out links to smoothen transition during start up. Certain control systems continue to use this link after startup.

PID blocks in modern control systems have status flags for data values. In this way a data value could be tagged with communication lost status. Some systems provide fail-safe mechanism by making use of this flag. For example, Foundation Fieldbus standard allows a limited number of communication failures, after which error is declared and the block enters failure state. This in turn forces the actual block mode to manual during communication failure. This approach alleviates the problem associated with communication loss but does not eliminate them. The proposed modifications to the PID to compensate for communication loss differs in that we explicitly address communication failures and take advantage of the related information.

Current control designs assume periodic samplings. However, this assumption does not hold in a wireless environment. In this paper, in order to cope with possible measurement lost, we propose a modified PID algorithm. The enhanced algorithm acts in the same manner as the standard PID algorithm when the communication is reliable. When it detects any communication lost, this algorithm can smoothen possible spikes in the output. We validate the new algorithm with several experiments in industrial DCS.

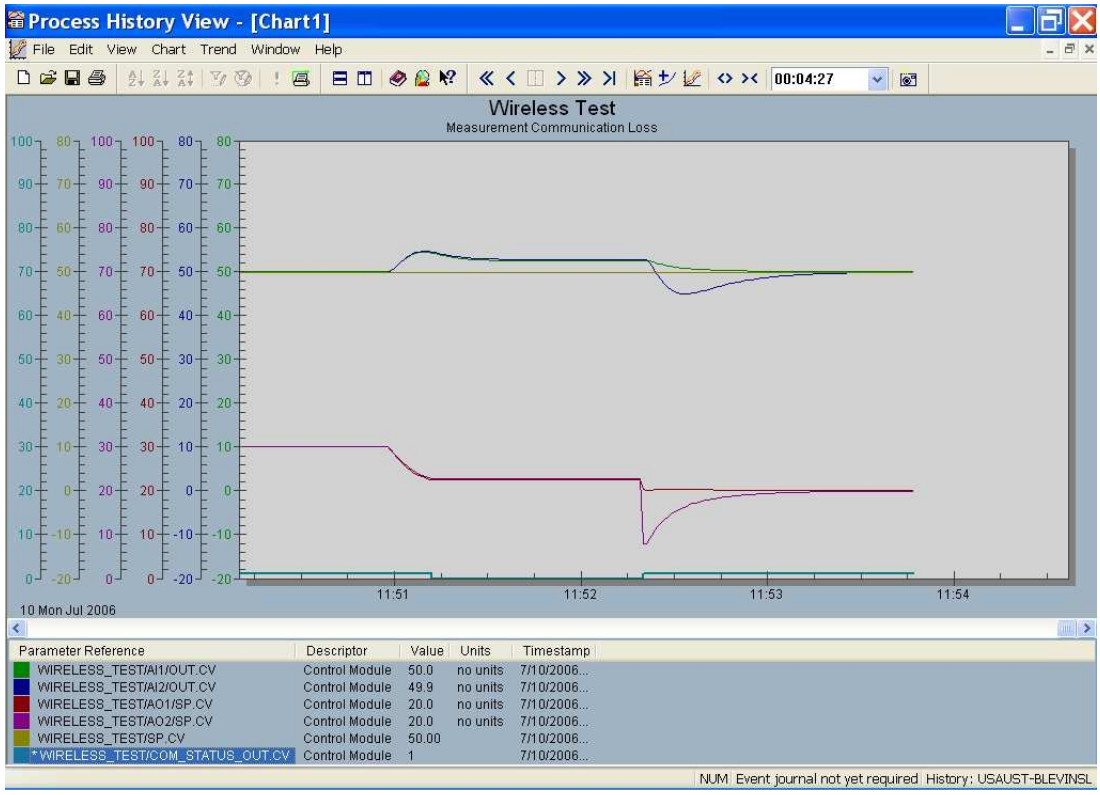


Figure 6.10: Missed Outputs with unmeasured disturbances

Chapter 7

Conclusion

7.1 Summary

Traditionally, schedulability analysis focuses on a scheduling model (e.g., the Lib & Lapland periodic task model [36]) and performs analysis (e.g., RAM) on input task sets which conform to the model [28, 37, 38, 40, 45]. This paradigm has proven effective in practice as long as the application can be abstracted to fit the scheduling model. In case of a misfit, the schedulability analysis tool cannot be used. In this dissertation, we follow a drastically different paradigm: it treats real-time scheduling as a model construction problem from the formal specification of timing constraints, specifically RTL (Real Time Logic) [31] formulas. We first translate the semantics of system requirements from different application domains into RTL formulas. The same basic synthesis engine is used to solve the resulting RTL formulas, with the long-term goal of incorporating more powerful results from real-time scheduling theory to be used as domain-specific heuristics for the synthesis engine.

In this dissertation, we extended the MSP.RTL in two broad directions: improving the tool itself and broadening the application domain of the tool to include wired and wireless industrial process control. For the tool itself, we

cut the memory usage of the tool in half. The new engine systematically used various techniques from the domain of constraint satisfaction programming, such as consistency checking, conflict-based backjumping, and backmarking. The resulting tool is faster and applicable to a broad application domains.

After experimenting the new tool with the traditional Boeing 777 AIMS problem, we applied it to wired and wireless industrial process control. Enhanced with domain specific heuristics, the tool was successfully applied to solve the block assignment problem in Fieldbus networks, where each block comprising the control system is assigned to a specific device such that certain metrics of the system can be optimized. We also experimented the tool to schedule communications on a simulated wireless industrial network.

In order to experiment with real wireless process control systems, we went ahead to build a prototype WirelessHART stack on a Freescale microcontroller toolkit. Through a carefully designed timer, we were able to implement synchronous and secure wireless communications within $10ms$ time slots. Several demonstration applications have been built on top of the WirelessHART stack. We are still actively developing the network manager.

Even with the scheduler tool to regulate communications in a wireless process control, there is still a non-neglectable possibility that a communication task misses its deadline. In order to handle this type of failures, we proposed to make the control modules aware of the unreliability of wireless links. PID (Proportional, Integral, Derivative) modules are the most used control modules. We developed PIDPlus, an enhanced PID algorithm to cope

with possible lost inputs and outputs. It has been shown that PIDPlus can drastically improve the stability of the control loop in cases of unreliable wireless communications.

7.2 Future Work

7.2.1 The MSP.RTL tool

Currently, MSP.RTL is an off-line scheduler and best suited to predictive models of constraints which are generally pretty accurate. When it is applied to wireless industrial process control, the underlying network is usually well planned and changes slowly so that the offline scheduler has time to respond to the changed requirements. Without any modification, the tool would have difficulty handling dynamic environments (for example, ad-hoc mobile networks), where tasks and resources change quickly. In this case, there would be no way to provide real time guarantee. The scheduling goal would be to minimize the changes in resulting schedule when the environment changes [46].

7.2.2 Network manager in WirelessHART networks

We have built a WirelessHART platform for wireless process control and are putting efforts to develop a general network manager for such networks. As mentioned in Chapter 5, a network manager in WirelessHART networks is a piece of very complicated software. First, it is responsible for routing the packets. Then, based on the routes, it can schedule the packets to meet system

timing requirements. Every routing change will affect the schedule. Just because of the relying on routing, a network manager also needs to minimize the changes in routing, which adds another complexity.

7.2.3 PIDPlus

We implemented PIDPlus in a commercial product and evaluated its effectiveness through simulations. It would be interesting to implement the algorithm in a WirelessHART controller and experiment the system in a real field. Combined with a WirelessHART network manager, it would be a complete solution for wireless process control.

Bibliography

[1] 1321xEVK Product Summary. www.freescale.com/webapp/sps/site/prod_summary.jsp?code=1321xEVK.

[2] Bluetooth. www.bluetooth.com/bluetooth/.

[3] IEEE 802.11 Task Group. http://grouper.ieee.org/groups/802/11/.

[4] IEEE 802.15.4 WPAN Task Group. www.ieee802.org/15/pub/TG4.html.

[5] Implementation of aes in c/c++ and assembler. http://fp.gladman.plus.com/cryptography_technology/rijndael/index.htm.

[6] FIPS Publication 197. *Advanced Encryption Standard (AES)*. U.S. DoC/NIST, November 2001.

[7] K. Akkaya and M. Younis. A survey of routing protocols in wireless sensor networks, 2005.

[8] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. In *IEEE Communication Magazine*, August 2002.

[9] K. Altisen, G. Gößler, A. Pnueli, J. Sifakis, S. Tripakis, and S. Yovine. A framework for scheduler synthesis. In *Proc. of IEEE Real-Time Systems Symposium(RTSS)*, pages 154–163, 1999.

[10] P. Bahl, R. Chandra, and J. Dunagan. Ssch: Slotted seeded channel hopping for capacity improvement in ieee 802.11 ad-hoc wireless networks. In *Proceedings of ACM MobiCom*, 2004.

[11] T. Blevins, G. McMillan, W. Wojsznis, and M. Brown. *Advanced Control Unleashed: Plant Performance Management for Optimum Benefit.* ISA Press, 2002.

[12] Dick Caro. *Wireless Networks for Industrial Automation.* ISA Press, 2004.

[13] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A Categorization of Real-Time Multiprocessor Scheduling Problems and Algorithms. In *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, pages 30–1–30–19. Chapman and Hall/CRC, 2004.

[14] T. Carpenter, K. Driscoll, K. Hoyme, and J. Carciofini. ARINC 659 Scheduling: Problem Definition. In *Proc. of IEEE Real-Time Systems Symposium(RTSS)*, pages 165–169, 1994.

[15] D. Chen, M. Nixon, T. Aneweer, R. Shepard, T. Blevins, G. McMillan, and A. Mok. Similarity-based traffic reduction to increase battery life in

a wireless process control network. In *ISA EXPO Technical Conference*, October 2005.

[16] S. Davari and S. Dhall. An on-line algorithm for real-time tasks allocation. In *Proc. of IEEE Real-Time Systems Symposium(RTSS)*, pages 194–200, 1986.

[17] R. Dechter and D. Frost. Backtracking algorithms for constraint satisfaction problems - a tutorial survey. Technical report, University of California at Irvine, 1998.

[18] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.

[19] M. Dworkin. *Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality.* NIST Special Publication 800-38C, May 2004.

[20] Jr. Edgar H. Callaway and Edgar H. Callaway. *Wireless Sensor Networks: Architectures and Protocols.* CRC Press, August 2003.

[21] Emerson Process Management. `http://www.easydeltav.com`.

[22] Shimon Even. *Graph Algorithms.* Computer Science Press, Inc., 1979.

[23] Foundation FieldBus. `http://www.fieldbus.org`.

[24] S. Ganeriwal, R. Kumar, and M. B. Srivastava. Timing-sync protocol for sensor networks. In *ACM Conference on Embedded Networked Sensor Systems (SENSYS 2003)*, Novemember 2003.

[25] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[26] S. Han, J. Song, X. Zhu, and A. Mok. Wi-htest: Compliance test suite for diagnosing devices in real-time wirelesshart networks. In *Proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium*, 2009.

[27] HART Communication. `http://www.hartcomm2.org/index.html`.

[28] R. Holte, A. Mok, L. Rosier, I. Tulchinsky, and D. Varvel. The pinwheel: A real-time scheduling problem. In *22th Hawaii International Conference on System Sciences*, January 1989.

[29] K. Hoyme and K. Driscoll. SAFEBus^{TM}. In *the 11th Digital Avionics Systems Conference*, pages 68–73, 1992.

[30] ISA100: Wireless Systems for Automation. `http://www.isa.org/MSTemplate.cfm?MicrositeID=1134&CommitteeID=6891`.

[31] F. Jahanian and A. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transaction on Software Engineering*, 12(9):890–904, 1986.

[32] F. Jahanian and A. Mok. A graph-theoretic approach for timing analysis and its implementation. *IEEE Transactions on Computers*, 36(8):961–975, 1987.

[33] John Shaw. `http://learncontrol.com/pid`.

[34] Raymond Barrett Jose A. Gutierrez, Edgar H. Callaway. *IEEE 802.15.4 Low-Rate Wireless Personal Area Networks: Enabling Wireless Sensor Networks*. IEEE, April 2003.

[35] Q. Li and D. Rus. Global Clock Synchronization in Sensor Networks. In *IEEE Infocom 2004*, March 2004.

[36] C. Lib and J. Lapland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of ACM*, 20(1), Jan 1973.

[37] J. Lib, L. Redondo, Z. Deng, T. Tia, W. Shih, and R. Bettati. PERTS: A prototyping environment for real-time systems. In *Proc. of IEEE Real-Time Systems Symposium(RTSS)*, pages 184–188, 1993.

[38] L. Luqi. Real-time constraints in a rapid prototyping language. *Computer Languages*, 18(2), 1993.

[39] A. Mok, D. Tsou, and R. Rooij. The MSP.RTL real-time scheduler synthesis tool. In *Proc. of IEEE Real-Time Systems Symposium(RTSS)*, pages 118–128, 1996.

[40] Aloysius K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*. PhD thesis, MIT, 1983.

[41] D. Oh and T. Baker. Utilization bounds for n-processor rate monotone scheduling with static processor assignment. *Real-time Systems*, 15(2):183–192, 1998.

[42] Y. Oh and S. Son. Allocating fixed-priority periodic tasks on multiprocessor systems. *Real-time Systems*, 9(3):207–239, 1995.

[43] PID controller. `http://en.wikipedia.org/wiki/PID_controller`.

[44] ProfiBus. `http://www.profibus.org`.

[45] K. Ramamritham. Allocation and scheduling of complex periodic tasks. In *Proc. of International Conference on Distributed Computing Systems(ICDCS)*, pages 108–115, 1990.

[46] S. Smith. Is scheduling a solved problem? In *First Multidisciplinary International Conference on Scheduling: Theory and Applications*, August 2003.

[47] J. Song, S. Han, A. Mok, D. Chen, M. Nixon, M. Lucas, and W. Pratt. WirelessHART: Applying Wireless Technology in Real-Time Industrial Process Control. In *Proc. of Real-Time Technology and Applications Symposium(RTAS)*, April 2008.

[48] J. Song, S. Han, A. K. Mok, D. Chen, M. Lucas, and M. Nixon. A study of process data transmission scheduling in wireless mesh networks. In *ISA EXPO Technical Conference*, October 2007.

[49] J. Song, S. Han, X. Zhu, A. Mok, D. Chen, and M. Nixon. A complete wirelesshart network. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 381–382, New York, NY, USA, 2008. ACM.

[50] J. Song, A. Mok, D. Chen, and M. Nixon. Using Real-Time Logic Synthesis Tool to Achieve Process Control over Wireless Sensor Networks. In *The 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, August 2006.

[51] J. Song, A. Mok, D. Chen, and M. Nixon. Optimizing Distributed Foundation Fieldbus Process Control with MSP.RTL Tool. In *The 5th IEEE International Conference on Industrial Informatics*, July 2007.

[52] J. Song, A. K. Mok, D. Chen, M. Nixon, T. Blevins, and W. Wojsznis. Improving pid control with unreliable communications. In *ISA EXPO Technical Conference*, 2006.

[53] J. Stankovic, T. Abdelzaher, C. Lu, L. Sha, and J. Hou. Real-time communication and coordination in embedded sensor networks, 2003.

[54] Wireless Industrial Networking Alliance. `http://www.wina.org`.

[55] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient mac protocol for wireless sensor networks, 2002.

[56] ZigBee Alliance. http://www.zigbee.org/en/index.asp.

Vita

Jianping Song was born in Jiangxi, a province in south China. He received his B.S. and M.S. degree in Computer Science in 1998 and 2001, respectively, both from Tsinghua University, Beijing, China. Then he got admitted to the Ph.D. program in the Department of Computer Science at the University of Texas at Austin in 2001. Thereafter he has been working on his Ph.D. degree under the advising of Professor Aloysius K. Mok. He obtained the degree of Master of Art in Computer Science in 2007 en route to his Ph.D. degree.

Permanent address: 2314 Wickersham Ln Apt 307, Austin TX 78741

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.