Memoryless subsystems

J. S. Fenton

University of Cambridge Computer Laboratory, Corn Exchange Street, Cambridge CB2 3QG

A memoryless subsystem is incapable of communicating unauthorised information about data input to the outside world. Such systems are important in the study of protection systems, but are difficult to implement. This paper derives a model of such a system and further gives a proof of its correctness.

(Received January 1973)

A memoryless subsystem is a program or procedure on a computer utility which is guaranteed to have kept no record of data supplied when it has completed its task. An oft quoted example of such a subsystem is an Income Tax program (Graham & Denning, 1972); to function it must have access to confidential data such as the customer's income and expenditure, but must not keep a record for anyone else to see later.

The usual approach to this problem is to encapsulate the tax program in an environment entirely controlled by the customer. This is often achieved by the use of stand-alone computer time where the machine is wiped clean at the end. This fails to solve the problem when the tax program requires protection; in this case the tax program and the customer's program are in a state of mutual suspicion.

It now becomes necessary to consider a computer system that allows the interaction of mutually suspicious subsystems. The fact that the suspicion is mutual and not total is important; two subsystems that are totally suspicious will never interact. This mutual suspicion indicates some level of trust; this is assumed to be invested in a set of shared supervisory procedures.

On an appropriate system it would be possible for the customer's program to encapsulate the tax program and inspect any output produced. In general this approach fails since there is no algorithm for deciding whether information is contained in an output stream; the tax program may use means such as the position of the date or the number of spaces on a line to convey information. Also the tax program may wish to send charging information and similar output to its author without customer intervention. It is a consequence of the problem that the charge may not depend on, for example, the customer's income.

A restatement of the general problem is as follows: a computer user wishes to run a proprietary subsystem which requires him to supply confidential or private data. The subsystem cannot be inspected and appears to the user as a black box. For input it requires several streams of data, some containing private and some non-private information. As output it generates several streams, some allowed to contain private information to be returned to the customer and some containing non-private information to be returned to the author (See **Fig. 1**).

The problem is to provide hardware support so that, despite the execution of an arbitary program over which the customer has no control, the normal output can in no way depend on private input.

It seems unlikely that a solution to this sort of problem can be provided by protection mechanisms currently studied (Needham, 1972). Such protection systems are concerned with the addresses available to a process irrespective of the contents of these addresses. The tax problem is concerned with the protection of data, irrespective of where it is stored. One might regard these ideas as orthogonal.

The solution is presented in terms of an abstract machine. This enables a proof of the correctness of the model to be given, something that is rarely possible for protection systems on practical machines.

An abstract computer model

The use of simplified computer models has previously been restricted to the study of computability. There have been many alternatives equivalent (computationally) to Turing's original machine but the model chosen because of its resemblance to practical computers is that devised by Minsky (1967). Minsky established that a sufficient computer needs only two (infinite) registers, a zero register and three instructions.

1. a' $\quad a := a + 1$
2. $a^-(n)$ $\quad$ if $a = \emptyset$ then goto n else $a := a - 1$ fi
3. halt $\quad$ stop and display output registers

Other instructions can be defined in terms of these. For example, if z is the zero register

$$z^-(n)$$

is an unconditional jump to n, denoted as go(n). Similarly the subroutine

$$\begin{array}{ll} n: & a^-(m) \\ & \text{go}(n) \\ m: & \end{array}$$

always exits with $a = \emptyset$. This is denoted as a^0. With these instructions one can write a program to simulate a universal Turing machine, thus establishing the computational power of the model. Note that output is not available until the processor halts.

Data marks

A data mark is a field of information associated with the basic

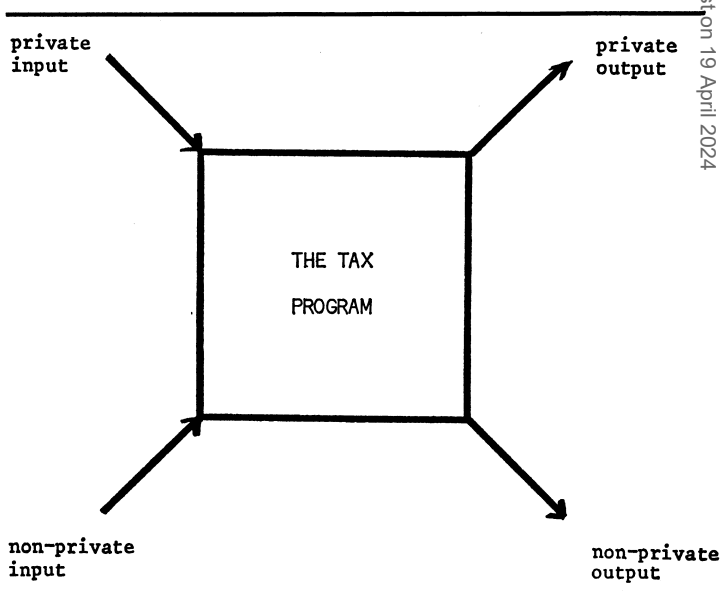

Fig. 1

unit of storage in a computer utility (e.g. a word, a byte, etc.). For the purpose of this document it will suffice to consider a two-valued data mark where the values are denoted as **null** and **priv.**

To protect information it would be logical to attach data marks to information and force the data mark to remain with that information wherever it may be stored. For reasons that will become clear later (see General Purpose Registers below) this cannot be done and a storage location has a fixed data mark. Thus, if a is a storage location, **a** is the data mark of a and is constant. Then

$$\mathbf{a} = \textbf{null} \text{ or } \textbf{priv}$$

Information extracted from a storage register has the data mark of that register attached. Information can only be stored in a register if the data mark of the register **a** and the data mark of the information **b** agree in the sense that

$$M(\mathbf{a}, \mathbf{b}) = F_{op}(\mathbf{a}, \mathbf{b})$$

where F_{op} is an operation dependent function and M is the interaction matrix:

M	**null**	**priv**
null	**null**	**priv**
priv	**priv**	**priv**

There are practical motivations for this approach. In a segmented machine it is usual to require that all words in a segment have the same protection status. It is also wasteful in memory to reserve one bit per word for a data mark if this is unnecessary. However, the real reason for this approach is that variable data marks do not appear to solve the problem completely as is demonstrated by an example below.

Information flow in the Minsky machine

In a Minsky machine all operations are on a single register and accordingly the definition of data marks in terms of binary operations may seem inappropriate. But consider the following program for the Minsky machine to add a to b.

```
1  n:   a⁻(m)
2       b'
3       go(n)
4  m:
```

It can be thought of in the following intuitive manner. Line 1 extracts a piece of information about the contents of a and line 2 stores that information in b. This is repeated until all the information in a has gone.

However, between line 1 and line 2 one could insert an arbitrarily complex piece of program. Yet throughout the whole of that program the bit of information removed from a could be added to b at any point, or duplicated in other registers at will. Thus, at line 2 the piece of information removed from a exists in the position of the program pointer (i.e. the instruction counter in a conventional machine).

Thus the position in the program contains a quantity of information and this must be marked like any other piece of information in the machine. Thus, if p is the program pointer then **p** is its data mark. An operation on register a depends on $M(\mathbf{a}, \mathbf{p})$.

However, suppose register a has $\mathbf{a} = \textbf{priv}$ then an $a^{-}(n)$ operation by the above argument must mark $\mathbf{p} = \textbf{priv}$. But there is now no mechanism to unset the mark (i.e. set $\mathbf{p} = \textbf{null}$) and all subsequent operations must be **priv** type. This is a consequence of the program pointer 'remembering' the information indefinitely. Thus to unset **p** it is necessary to destroy all information in the program pointer.

There is only one place where the information extracted by a test could be lost and that is by returning to the instruction directly after the conditional. This leads to the concept of a linked conditional test instruction $a^{*}(n)$. This is an $a^{-}(n)$ instruction which, if the test succeeds, stacks a link (containing p and **p**) and jumps to n. There is a 'return' instruction that unstacks p and **p** thus resetting the processor mark.

It should now be possible for the program to test privileged information without loss of security. At some points it may go into a **priv** routine where all operations are restricted to **priv** data; at some point it must 'return'. The **null** path (the path through the program while $\mathbf{p} = \textbf{null}$) then followed cannot depend on what happened in the routine since this only affected **priv** data and thus cannot depend on whether the routine was entered or not. Thus it should be possible to perform general computations and guarantee that **priv** information cannot be moved to a **null** register.

The modified machine is constructed by adding the $a^{*}(n)$ and 'return' instructions to the Minsky machine repertoire. Additionally, every register on the machine is assigned a fixed data mark. The only variable mark is **p**. Using the concepts of information flow in a program as described above it is possible to derive the following instruction set, where x is any register

x'	if $\mathbf{x} = M(\mathbf{x}, \mathbf{p})$ then $x := x + 1$ fi
$x^{-}(n)$	if $x = \emptyset$ then (if $\mathbf{p} = M(\mathbf{x}, \mathbf{p})$ then goto n fi) else (if $\mathbf{x} = M(\mathbf{x}, \mathbf{p})$ then $x := x - 1$ fi) fi
$x^{*}(n)$	if $x = \emptyset$ then (*stack*$(p, \mathbf{p})$; $\mathbf{p} := M(\mathbf{x}, \mathbf{p})$; goto n) else (if $\mathbf{x} = M(\mathbf{x}, \mathbf{p})$ then $x := x - 1$ fi) fi
Return	$p, \mathbf{p} :=$ *unstack*()
Halt	if $\mathbf{p} = \textbf{null}$ then *halt* fi

Note that the Return instruction is assumed to have no effect if the stack is empty. Obviously the user must not be able to access the stack directly.

Requirements of a solution

Examining the concept of a data mark it becomes clear that a necessary and sufficient condition for the tax program to become 'memoryless' is that no bit of information marked **priv** can be copied into another register and marked **null.** When this is true a system is said to be *secure.* A 'bit' is taken to mean the smallest amount of information; for example the fact that the contents of register a is greater than the contents of register b.

The inability to move information about without losing **priv** marks is called 'retaining the mark'. In its most general sense it means that any (sequence of) operations cannot change the marks of data other than as described by the matrix M.

An important consequence of the universality of the model is that only binary values need be considered. It is possible (Minsky, 1967) to write subroutines $P(a)$ and $H(a)$ that give the parity and half of a respectively so that $a = P(a) + 2.H(a)$. Since it is possible to code any word in this manner then if it is possible to copy a binary value without retaining the mark then it is possible to copy a whole word by binary encoding and copying a bit at a time.

In the particular case under consideration the only important data is **priv** data; an operation retains the mark if it does not transfer information from a **priv** state to a **null** state. This leads to the following theorem.

Theorem 1

The system is secure if and only if the **null** path of the program cannot depend on any **priv** information.

For suppose the **null** path can depend on priv information. This implies that the program can follow two distinct branches depending on **priv** information and thus a binary value may be copied. By the above remarks any word may be copied and the system is not secure.

Conversely, if the system is not secure **priv** information can be

copied to a **null** word. Since the model is a universal computer, its path can depend on the copied information and thus on **priv** information.
Q.e.d.

The protected model

To provide any protection at all it is necessary to restrict the capabilities of the Minsky machine and it is clearly the case that the ordinary Minsky machine can perform data transformations that the protected model cannot. However, the residual machine must still be a universal computer on some subset of its registers or it will have become too degraded to be of use.

That the machine is a universal computer can be verified quite simply. One can define two submachines M_{null} and M_{priv} which are the protected model restricted to the **null** registers only and the **priv** registers and z only respectively. Then both these machines are universal machines provided that both contain at least two registers (excluding z). This follows from examining the instructions a' and $a^-(n)$ verifying that they form a Minsky machine on the relevant subset of registers while **p** is in the correct mode.

The following theorems prove that the protected Minsky model is secure up to a maximum limit and that a more secure system does not exist.

Theorem 2

While **p** = **priv** there is no way of altering a register x for which **x** = **null.**

The proof is by exhaustion of all possibilities. The 'Return' instruction does not alter any registers. Thus only the x', $x^-(n)$ and $x^*(n)$ instructions need be considered.

x' By definition this instruction reads

$$\underline{\text{if}}\ \mathbf{x} = M(\mathbf{x}, \mathbf{p})\ \underline{\text{then}}\ x := x + 1\ \underline{\text{fi}}$$

but, by hypothesis

$$\begin{aligned} M(\mathbf{x}, \mathbf{p}) &= M(\mathbf{null}, \mathbf{priv}) \\ &= \mathbf{priv} \\ &\neq \mathbf{x} \end{aligned}$$

so the instruction has no effect

$x^-(n)$ From the definition there are two possible cases

(*a*) $x = \emptyset$

but $M(\mathbf{x}, \mathbf{p}) = \mathbf{priv} = \mathbf{p}$ so that the program jumps to n. However, x is not changed

(*b*) $x > \emptyset$

As for the x' instruction

$$M(\mathbf{x}, \mathbf{p}) \neq \mathbf{x}$$

so the instruction has no effect

$x^*(n)$ As for the $x^-(n)$ instruction there are two possible cases

(*a*) $x = \emptyset$

This jumps to n irrespective of **x** and **p**. But x is not changed.

(*b*) $x > \emptyset$

Again $\mathbf{x} \neq M(\mathbf{x}, \mathbf{p})$ so the instruction has no effect.

Q.e.d.

The symmetric converse of this theorem is untrue. For if **p** = **null** and y is a register with **y** = **priv** the instruction y' alters the contents of y for

$$\begin{aligned} M(\mathbf{y}, \mathbf{p}) &= M(\mathbf{priv}, \mathbf{null}) \\ &= \mathbf{priv} \\ &= \mathbf{y} \end{aligned}$$

However, the corresponding theorem is:

Theorem 3

If **p** = **null** no change of path (i.e. jump) can take place that depends on **priv** information without setting **p** = **priv.**

Again, the proof is by exhaustion. Only the two instructions that cause a conditional jump need be considered. The only case when a jump can occur is when the register is zero.

Let y be a register with $y = \emptyset$ and **y** = **priv.** There are two cases corresponding to the two conditional instructions.

1. $y^-(n)$ Now if **p** = **null** then

$$\begin{aligned} M(\mathbf{y}, \mathbf{p}) &= M(\mathbf{priv}, \mathbf{null}) \\ &= \mathbf{priv} \\ &\neq \mathbf{p} \end{aligned}$$

So by definition this does not jump

2. $y^*(n)$ This always jumps when $y = \emptyset$ irrespective of **p** and **y**

However, if **p** = **null**

$$\begin{aligned} \mathbf{p} &:= M(\mathbf{y}, \mathbf{p}) \\ &:= M(\mathbf{priv}, \mathbf{null}) \\ &:= \mathbf{priv} \end{aligned}$$

Q.e.d.

The theorems are of interest in relation to the concept of information flow mentioned above. Theorem 2 states that the machine does not allow information to flow from **priv** to **null** although the reverse is true. Theorem 3 states that the machine does not allow **priv** information to be extracted while the processor state is **null.** These results are exactly what one would expect from the heuristic argument.

Theorem 4

If **p** = **priv** then the only way in which **p** can be reset to **null** is by a Return instruction.

Again, inspection of the instruction set reveals that only the $a^*(n)$ and Return instructions alter **p.** Now if **p** = **priv** then an $a^*(n)$ instruction with $a = \emptyset$ will set **p** by

$$\begin{aligned} \mathbf{p} &= M(\mathbf{p}, \mathbf{a}) \\ &= \mathbf{priv} \end{aligned}$$

for either value of a. Thus **p** cannot be reset to **null** except by a Return.
Q.e.d.

Theorem 5

Suppose the machine started with **p** = **null** and that it has halted. Then the system is secure.

For the program's **null** path cannot depend on any **priv** information. While in priv state, it cannot change any **null** register by Theorem 2. While in **null** state it cannot sense any **priv** information without moving into **priv** state by Theorem 3. But, by hypothesis, it has halted and is in **null** state; hence it must have 'returned' by Theorem 4. After a return it must continue executing the program immediately after the $a^*(n)$ instruction and thus continue along the **null** path. But, since no **null** registers have been changed this must be independent of entry into the **priv** routine. Hence the model is secure by Theorem 1.
Q.e.d.

The halting problem

The hypothesis of Theorem 5 requires that the program halts. However, the author can arrange that his program does not halt under certain **priv** conditions; for example, if **a** = **priv**

$$a^*(n) \qquad n\text{: go}(n)$$

will loop indefinitely only if $a = \emptyset$.

In practice this has a serious effect. The system can be considered as in **Fig. 2.**

If the author can sit waiting for output (this is not available until the machine halts) he may be able to observe the non-

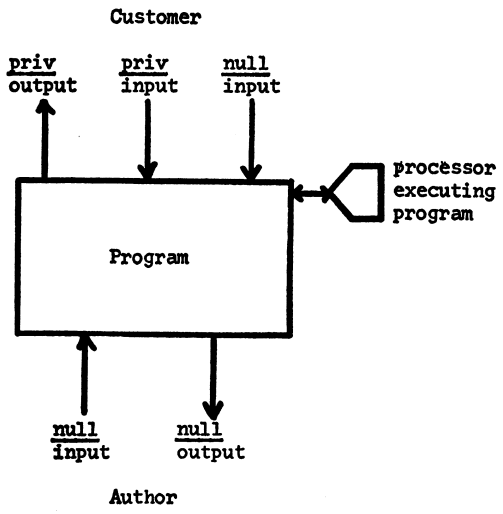

Fig. 2

halting of his program. This assumes that the author can estimate, ignoring the possibility of programming errors, a maximum bound on the program's computation time. In this way he could extract one bit of information.

Unfortunately, if the **priv** machine is to have universal computing power then it is impossible to improve on this. For the program is an arbitrary algorithm supplied by the author. If the hardware design could provide more security then it could determine whether the program will halt. But this contradicts the halting theorem for universal machines.

A simplification

A simplification of the model is possible in this particular case and may be of use in a practical design. Effectively the machine exists in two states, **priv** and **null**; it should be possible to provide a direct switch between them.

If the $a^*(n)$ instruction was replaced by an enter(n) instruction (that saved p, jumped to n and set $\mathbf{p} = \mathbf{priv}$) and the return instruction retained to restore p and set $\mathbf{p} = \mathbf{null}$ then the code

enter(n) n: $a^-(m)$
Return
m:

is identical to the effect of $a^*(m)$. The enter(n) instruction is equivalent to an $a^*(n)$ where $a = \emptyset$ and $\mathbf{a} = \mathbf{priv}$.

Thus these instructions are equivalent and can be defined in terms of one another. The simpler link instruction was not adopted because it is a special case, taking advantage of the bi-valued matrix M. The $a^*(n)$ instruction can be extended for any multi-valued matrix M and its properties are defined entirely by M. It therefore provides a general solution that should be applicable to other cases.

The general purpose register problem

The majority of practical computers have some general registers. From the use of fixed data marks it might seem that these registers would have to be pre-allocated to **priv** or **null** data. This would mean, for example, that programs would have to be re-written to act on **priv** and **null** data, an unsatisfactory state of affairs.

A seemingly more sensible approach to the problem would be to allow variable data marks; that is the data mark of a result is determined by the data marks of the operands, i.e.

$$a := b + c$$
$$\mathbf{a} := M(\mathbf{b}, \mathbf{c})$$

This would give rise to the following instruction set

a'	$a := a + 1$; $\mathbf{a} := M(\mathbf{a}, \mathbf{p})$
$a^-(n)$	if $a = \emptyset$ then (if $\mathbf{p} = M(\mathbf{a}, \mathbf{p})$ then goto n fi) else ($a := a - 1$; $\mathbf{a} := M(\mathbf{a}, \mathbf{p})$)fi
$a^*(n)$	if $a = \emptyset$ then ($stack(p, \mathbf{p})$; $\mathbf{p} := M(\mathbf{a}, \mathbf{p})$; goto n) else ($a := a - 1$; $\mathbf{a} := M(\mathbf{a}, \mathbf{p})$) fi
Return	p, $\mathbf{p}$:= unstack()
Halt	if $\mathbf{p} = \mathbf{null}$ then *halt* fi

Using this instruction set it is possible to demonstrate that the model *is not secure*. Consider the following program where register a contains $\emptyset$ or 1 and $\mathbf{a} = \mathbf{priv}$. Registers b and c are free with both **b** and **c** set to **null**. Initially $\mathbf{p} = \mathbf{null}$.

1		b^0
2		c^0
3		$a^*(n)$
4		$c^*(m)$
		⋮
10	n:	c'
11		Return
12	m:	b'
13		Return

Examination of the above program will show that there are only two control paths through the program represented by the following two tables:

1. $a = \emptyset$

		path	p	a	a	b	b	c	c
	1	b^0	null	$\emptyset$	priv	$\emptyset$	null	?	?
	2	c^0	null	$\emptyset$	priv	$\emptyset$	null	$\emptyset$	null
	3	$a^*(n)$	null	$\emptyset$	priv	$\emptyset$	null	$\emptyset$	null
n:	10	c'	priv	$\emptyset$	priv	$\emptyset$	null	1	priv
	11	R	priv	$\emptyset$	priv	$\emptyset$	null	1	priv
	4	$c^*(m)$	null	$\emptyset$	priv	$\emptyset$	null	$\emptyset$	priv

The exit condition is $b = \emptyset$; $\mathbf{b} = \mathbf{null}$

2. $a = 1$

		path	p	a	a	b	b	c	c
	1	b^0	null	1	priv	$\emptyset$	null	?	?
	2	c^0	null	1	priv	$\emptyset$	null	$\emptyset$	null
	3	$a^*(n)$	null	$\emptyset$	priv	$\emptyset$	null	$\emptyset$	null
	4	$c^*(m)$	null	$\emptyset$	priv	$\emptyset$	null	$\emptyset$	null
m:	12	b'	null	$\emptyset$	priv	1	null	$\emptyset$	null
	13	R	null	$\emptyset$	priv	1	null	$\emptyset$	null

Here the exit condition is $b = 1$, $\mathbf{b} = \mathbf{null}$.

Thus the program has copied a to b and removed the mark. So the system is not secure. This counter-example does not work in the case of fixed data marks for if $\mathbf{c} = \mathbf{null}$ always, then the c' instruction at n (line 10) would have no effect, so that the test at line 4 would always succeed leaving $b = 1$ in both cases.

It is not clear from the above example whether the failure is due to variable data marks or the instruction set chosen. However, Theorem 2 does not hold for variable data marks since it is now possible to alter a null register from **priv** state, although the result is marked **priv**. Hence, in the current framework, variable data marks are unsatisfactory.

Computation limitation

The section on halting suggests that total security is impossible. But consider the machine augmented with two counters i_n and i_p. Every instruction has the effect of decrementing either i_n if $\mathbf{p} = \mathbf{null}$ or i_p if $\mathbf{p} = \mathbf{priv}$. When i_p is zero a 'return' is forced and when i_n is zero the machine halts. Attempts to set $\mathbf{p} = \mathbf{priv}$ will fail if $i_p = \emptyset$. **Priv** and **null** modes 'tick' independently.

If a halt is forced it must occur while $\mathbf{p} = \mathbf{null}$ since i_n is only decremented in this case. However, now the customer can be certain that the program will halt within time t where

$$t = i_n + i_p$$

and so this system is totally secure by Theorem 5.

On casual inspection this may seem to contradict the remarks on halting. However, the modified **priv** machine is not universal and thus the halting theorem no longer applies. (The extra security comes from the customer having a bound on the computation time which was previously known only to the author). In practice this does not matter since the values of i_n and i_p can be set so that the program has sufficient time to complete its task.

These registers correspond to clocks in real machines. Note that the value of i_p must only be accessible while the processor is marked **priv** or **priv** information can be extracted. Nor must the computation time be accessible externally.

Acknowledgements

I would like to thank Dr. R. M. Needham for his constant help and encouragement throughout the development of these ideas. I am also grateful to many members of the Computer Laboratory for comments and discussions, in particular Professor M. V. Wilkes for reading and commenting on the paper, Dr. J. K. M. Moody for checking the theorems and Mr. W. D. Manville for several useful comments including a discussion on the general register problem.

References

GRAHAM, G. S., and DENNING, P. J. (1972). Protection—Principles and Practice, *AFIPS conference publications*, Vol. 42, p. 417.

MINSKY, M. L. (1967). *Computation; Finite and Infinite Machines*, Prentice-Hall.

NEEDHAM, R. M. (1972). Protection Systems and Protection Implementations, *AFIPS conference publications*, Vol. 41, p. 571.

Book reviews

Computer programs for computational assistance in the study of linear control theory, by J. L. Melsa and S. K. Jones, 1973; 198 pages. (*McGraw Hill*, £1·95, Second edition)

The stated goal of the book is to provide a set of computer codes which solve many of the computational problems of linear control theory. The book achieves this objective, at least so far as is possible in 198 pages. The volume provides a valuable basic set of routines for control system design and would be very useful to anyone wishing to try these methods in practice.

Programs are contained in the book for the following problems: time response computation, sensitivity analysis, modal control, observer design, series compensation, solution of Riccati equations, decoupling, frequency response, root locus and partial fraction expansions. The second edition of the book contains a number of useful additional features which were not in the earlier version. Five new design programs have been added and there is a new chapter containing worked solutions to some typical design problems.

The routines all appear to work according to specification and there are few remaining program errors. The input and output data format has been carefully explained and there should be no difficulty getting the programs to work even for someone with limited computer knowledge. For anyone wishing to modify or improve the programs, it is unfortunate that few comments have been included and that there are no flowcharts. However, this extra information would have undoubtably increased the cost of the book without helping the average reader.

The main limitation of the programs is that they are restricted to low order systems. This is largely a consequence of the fact that, in every case where a choice existed, the authors favoured a simple algorithm rather than a complicated one with superior numerical properties. However, the programs will undoubtedly provide satisfactory solutions in simple cases and in more complex situations the programs at least provide a useful first step before resorting to the more specialised techniques.

The book is highly recommended to both teachers of control theory and practising engineers. At £1·95 the book is extremely good value for money.

G. C. GOODWIN (London)

Short notice

Computer Applications and Facilities for Science and Technology in the Asian and Pacific Region

This is the fourth in a series of directories published by the Registry of Scientific and Technical Services, Jamieson ACT, Australia, whose aim is to encourage cooperation between scientific and technical groups working in agriculture, forestry, fisheries, engineering, mining, industry, computing and other fields associated with development in the countries agreeing to be included in the directories.

The directory is in three parts, the first containing information on computer installations, the second on computer users, and the third being two indexes: a subject index to part 2 and an organisational index to all the groups listed.

The countries taking part are Australia, China (Taiwan), Japan, Korea, Malaysia, New Zealand, The Philippines and Thailand. Organisations are listed alphabetically within subdivisions of Government, University and private sector, under the country.

Interesting facts emerge on perusing this directory, such as that the biggest staffs are employed by the Victoria University of Wellington, New Zealand, with 700 professionals providing a computing service and training to students and graduates, the Taiwan Sugar Corporation, with 3,301 professionals in a total of 6,984 to provide a general computing service and data acquisition, and the Taiwan Ministry of Communications, with a staff of 10,230 of which 8,318 are professional, giving a computing service and administrative training.

At the other end of the scale, the Australian Government has several departments run with two professional staff out of a total of two, some university departments in Australia, New Zealand and Japan have only two staff, and an engineers computer bureau in Wellington has a staff of only two professionals to provide a complete computing service for engineers.

The directory, priced at A$10.00, is compiled and published by the Registry of Scientific and Technical Services and distributed by the Queensland University Press, St Lucia, Queensland, and the Australian Government Publishing Service, Canberra, Australia.